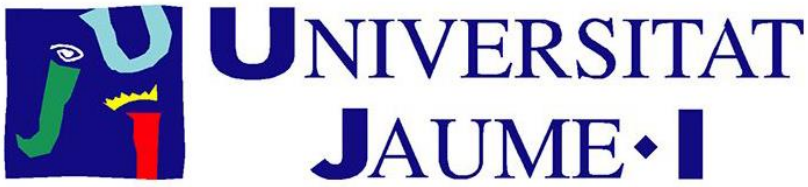
Development of procedural cities generation in Unity3D using Houdini



Videogame Design and Development Degree

Final Degree Work report

Author: Pablo Garcia Segarra

Bachelor's Degree in Video Game Design and Development

Universitat Jaume I

June 3, 2019

Advisor: José Ribelles Miguel

ABSTRACT

This document reviews the process of generating a 3D city procedurally using the software SideFX Houdini and implementing the generation into real-time engines. The main objective is to obtain a layout, procedurally or from an image, containing different buildings, streetlights, sidewalks, etc. The main interest regarding this generation is to be optimized in order to be ready-to-use inside the videogame production workflow.

CONTENTS

LIST OF FIGURES

1. TECHNICAL PROPOSAL

This chapter presents an overview of the project. The motivation of the project, a list of the principal objectives to achieve with work done, a planning of this development and the expected results are also explained in the following sections.

1.1. SUMMARY

This point presents the technical proposal for the final degree project in the Degree in Design and Development of Videogames. The work will consist in developing in Houdini and implementing in Unity3D a tool to generate cities (houses, buildings, streets, etc.) in a procedural manner, following established parameters.

The use of this tool will be possible both in the field of video games and in the field of architecture.

In videogames, the objective will be to speed up the environment creation. It would allow generating a fictional city from scratch, in a very fast way and ready to be used as a 3D environment.

As for architecture, its usefulness lies in, given a terrain, being able to carry out a simulation with very little cost of the result that we would obtain if that terrain were urbanized.

1.2. INTRODUCTION AND MOTIVATION OF THE PROJECT

The work will be based on learning the use of the Houdini tool. For this purpose, I will set the objective of generating a city in a procedural manner that varies according to the parameters desired by the future user, such as the number of buildings, type of buildings, type of streets, the width of streets, etc.

This tool will be available for both Houdini and Unity3D, in order to generate the base mesh to use it in a 3D Software for architectural purposes or, import it directly into unity for Games.

To do this, I will establish four main milestones.

The first is to develop a tool in Houdini that, by establishing some parameters, will generate a building for you.

These buildings will have geometry, colliders, and UVs for further texturing. Then a tool will be developed that, given a delimited area and parameters, generates a system of streets, trying to distribute them in the most plausible and realistic way possible.

After the completion of these tools, I will proceed to develop the master tool that, combining the two previous ones, generates a realistic city following the established parameters.

Finally, these developed tools will be implemented into Unity3D for using them directly from there.

Regarding the motivations, one of the main reasons for the choice of this topic is that I wanted to develop a topic that relates the programming and modelling knowledge learned during the degree in a different and innovative way.

In the second place, the main motivations for basing the end-of-degree project on the use of Houdini is that this program is becoming increasingly popular within the industry, such as videogames, architecture, visual effects, films, etc.

Moreover, the number of jobs that require knowledge of this program is currently growing fast. Therefore, its learning during the realization of this project is an investment in the future that can open some doors for job searching.

1.3. RELATED SUBJECTS

- VJ1212 - Graphics Communication
- VJ1216 - 3D Design
- VJ1223 - Video Game Art
- VJ1221 - Computer Graphics

1.4. OBJECTIVES

- Create a tool in Houdini to generate procedurally a house model, based on the parameters established.
- Create a tool in Houdini to generate procedurally roads in an area, using the given parameters.
- Create a 3D city generation tool, which uses the procedural houses and roads to create a city by using the given area and parameters.
- Allow the option to export the City/House/Road mesh directly into a 3D format or create it within the Game Engine.
- Integrate this tool inside Unity, to use it directly from the editor.

1.5. PROJECT SCHEDULE

This section describes the division of project's tasks and its estimate of the required time in hours.

Task	Estimated duration (in hours)
Documentation	
Technical proposal	8
Final report	50
Project video	4
Project defence preparation	12
Analysis	
Analysis and design document	24
Houdini introduction	20
References	6
Development	
Streets procedural generation	30
Buildings procedural generation	40
City procedural generation	60
Houdini plugin integration	12
Design	
Textures and shaders	8
Unity scene set-up	6
Testing	
Export result to Unity	20
Total	300

1.6. EXPECTED RESULTS

The resulting Project is expected to:

- Generate procedurally houses within given parameters (type, height, width, depth, floors, windows…).
- Generate procedurally roads within given parameters and designated area.
- Generate a city procedurally by using the roads and houses already generated to adapt to the given parameters and available terrain area.
- Assign the desired city parameters in Houdini to generate and export the city as a mesh.
- Assign the desired city, roads and houses parameters in Unity through a script and generate the city inside the engine.
- Generate a 3D city model with textures, ready to be used (Game Engine or any 3D Tool).
- Be easily scalable for future expansion.
- All models generated will be able to be exported as mesh, using the principal mesh formats (FBX, obj…).
- All models will have UVs unwrapped to further texturing using tileable textures or in Substance Painter.

1.7. TOOLS

- Unity 3D 2018

- SideFX Houdini & Houdini Engine
- Microsoft Word
- Visual Studio 2017
- GitHub
- Substance Designer
- Photoshop

2. DESIGN

This chapter discusses the Design of the project, which outlines the implementation description, tools and technologies used along the project.

2.1. INTRODUCTION

2.1.1. CONCEPT

This Project is based on the development of procedural generation tools to optimize the creation of virtual environments workflow.

Mainly, is about procedural cities generation, a process that takes a lot of time if done through traditional modelling techniques such as box modelling.

With the creation of this tool, a lot of time can be saved, specifically in the first steps of the project, on prototyping and blockout phases.

2.1.2. MAIN FEATURES

This tool is expected to, using the user input, obtain the desired models that fit the values established.

The behaviour of the tool is different in Unity compared to the source tool in Houdini.

In Unity, the final result of the execution of the tool is a model prefab, made of other small prefabs (Each building will be a prefab) that are ready-to-use in the Engine.

On other way, in Houdini, the result of the execution is a Mesh stored in some standard format, such .FBX.

Both types of model generation have some parameters that will not vary. They both will have UV coordinates and Material ID attributes, in order to easily be textured after (or during) generation by applying tileable textures.

2.1.3. TARGET AUDIENCE

The main reason why the environment selected is a city urban area against any other type of model is because its large usage within different areas, such as videogames industry or any other.

Mainly, in the videogame industry, this generation tool has a great utility thanks to its fast and easy capability to generate a virtual environment from scratch.

Moreover, more industries can be benefited from using this tool. One of them is the architecture and urbanism sector. They obtain in a fast way a preview of the result of the urbanization process of a delimitated area.

Finally, there is another industry that benefit of using it. This industry is the development of AIs for autonomous driving vehicles, allowing them to generate close to infinite environments to train the AI to drive.

2.2. VISUAL STYLE

Visual realism is one of the main components of the project. Therefore, this section will be strongly emphasized during its development.

2.2.1. BUILDINGS

For the modelling of buildings, they will be based mainly on apartment blocks, since they take advantage of the benefits of procedural modelling, allowing greater flexibility.

Image 1: Building reference style virtual

These apartments are based in the New York architecture.

Image 2: Building reference style real

In any case, it is not excluded to include a greater variety of buildings to generate diversity.

2.2.2. ROADS

As for the style of the streets, they do not have a greater complication, it will simply depend on the quality of the textures and the adequacy of the UV coordinates to avoid seams and stretches.

In relation to the road's distribution, they will follow real world references. For instance, this zone from New York, will be used as reference.

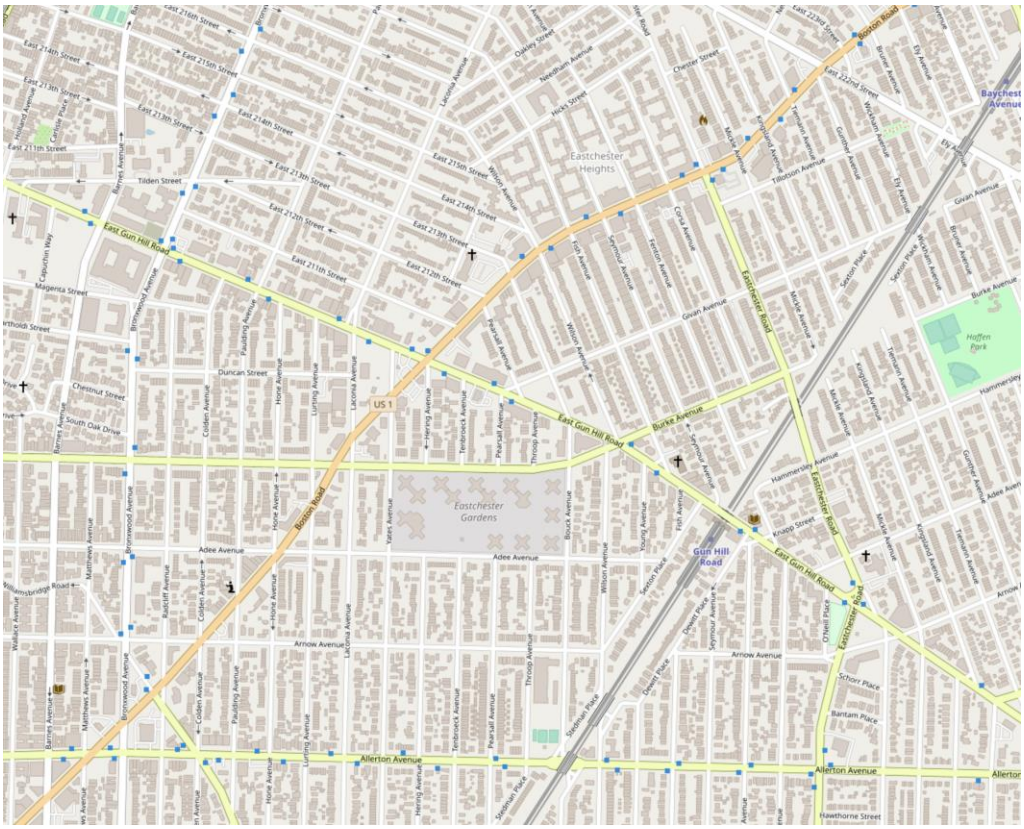


Image 3: Reference city layout map

2.2.3. TEXTURES

For the use of textures in the generated models, is going to be expected most of the textures used are generated by myself from scratch.

However, the use of open source textures is not ruled out in order to speed up the development of the rest of the parts, since the generation of textures is a time-consuming task.

2.3. PROCEDURAL MODELLING

2.3.1. BENEFITS

The reason of choosing procedural modelling is that it has advantages over traditional modelling, especially in models with parametrical features such as buildings or hard surface models. The principal benefits of procedural modelling are:

- Non-destructive modelling
- Ready-to-use assets
- Infinite variations
- Increases production speed
- Tweakable parameters
- Avoidance of repetitive tasks

2.3.2. TWEAKABLE PARAMETERS

In order to generate the final model, the user will have some parameters to modify, obtaining the result that fits the most to their criteria. Some of the parameters are:

2.3.2.1. BUILDING

- Floor Height
- Number of floors
- Window size
- Window Ledge size
- Door Size
- Materials
- Type of roof

2.3.2.2. CITY

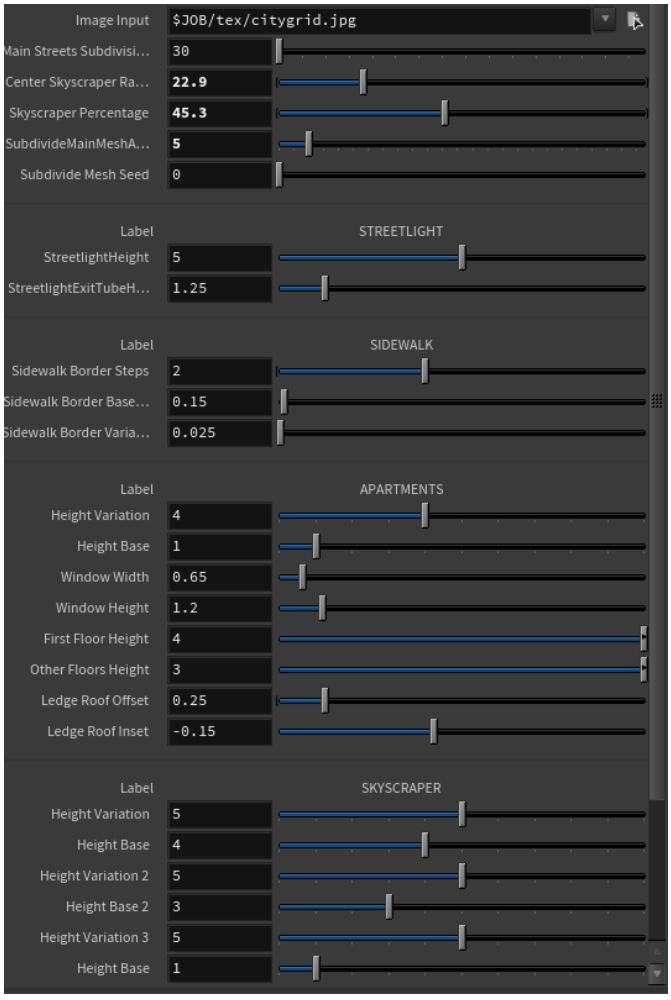


Image 4: City asset external parameters

Some parameters will be overridden when generating the city, with the purpose of optimization.

For instance, when creating the city, this script will randomize the offsets and values to get the best adaptation to the area. It should be noted that different parameters could be used during the development phase of the project.

2.3.3. ROADS GENERATION

For road generation, following some of real-world patterns and topography, the process is divided in two parts. The first one consists in generating the primary boundary roads. Then, in a second iteration, the secondary or pedestrian roads are created, covering each group.

2.3.4. BUILDING GENERATION

For the building's generation, the base process is going to be divided into different parts.

In the first part, we need to establish a function that randomly generates the shape of the base of the house.

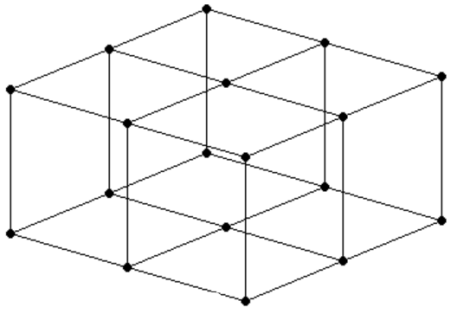


Image 5: Building base generation concept

This will be our base, from which everything else of the building will be generated.

From this base, we will generate the walls, extruding the external edges upwards the height of a single floor, so that each wall has an established height. Then walls are duplicated and applied a translation offset along the Y-axis, the same times as the number of floors indicated.

At this point, we will have a solid building block.

Now we will proceed to create windows and a main door holes by using Booleans with its shape.

Finally, the final window or door model will be placed in each corresponding hole.

2.4. SOFTWARE INTEGRATION

As we know, the main objective is the implementation in Unity of the tool created in Houdini. To do this, we must consider the software required for proper integration with the engine.

On the one hand, Houdini software is needed for the creation and development of the required tools.

On the other hand, you need Houdini Engine, which is the software necessary for the execution of the tools generated in Houdini directly within any game engine (Unity, Unreal...) or 3D software (Maya, 3ds Max...). This is how the corresponding bridge will look like.

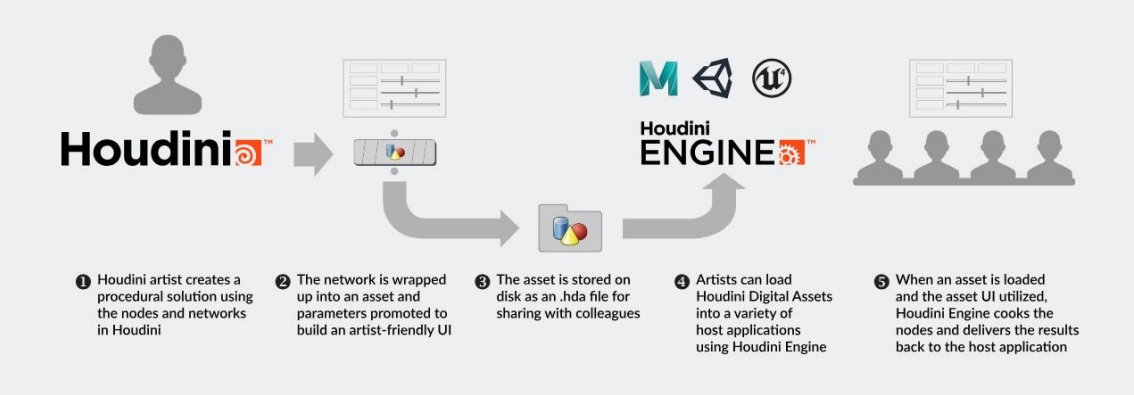


Image 6: Houdini Pipeline Roadmap

It should be noted that Houdini and Houdini Engine are independent of each other; therefore, a different license is required for each one.

Finally, in this project, the integration of the software and the interaction with the user would follow this flow chart.

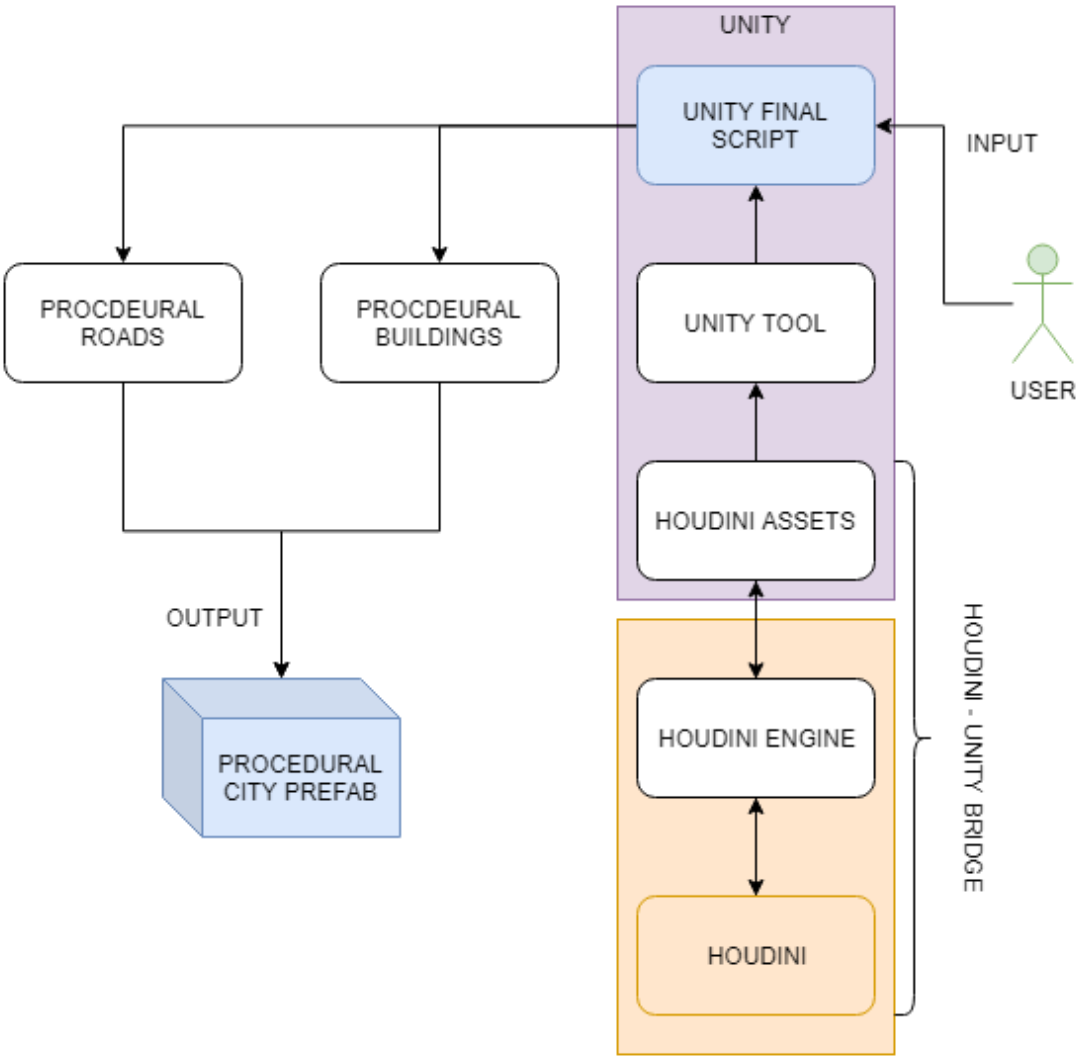


Image 7: Asset integration flowchart

2.5. SOFTWARE PLATFORMS

One of the most favourable advantages offered by Houdini is its wide capacity to export the tools created in the program itself to other programs, whether they are game engines or 3D.

The development of this Project will focus mainly on the implementation of the tool in Unity3D.

However, thanks to the flexibility offered by Houdini Engine, this tool will also be able to be used, without any necessary modification, by other software such as Unreal Engine, Maya or 3Ds Max.

3. WORK DEVELOPMENT

This chapter discusses all the procedures, decisions and implementations taken during the development process, and the results of this decisions.

3.1 APARTMENT BUILDING GENERATION

3.1.1 BUILD-UP BASE STRUCTURE

There are two types of generation for the base of the apartment.

The first one receives an input geometry and modifies it in order to start building from this base. This input is mostly used by the city generation asset, as it sends the geometry area and this asset returns the apartments.

The second one, is more likely to be used for this asset standalone. The base geometry is generated by a curve (having more than 3 points), and then the geometry is also adapted to generate the full structure. This curve is created by the user, just adding points to a surface.

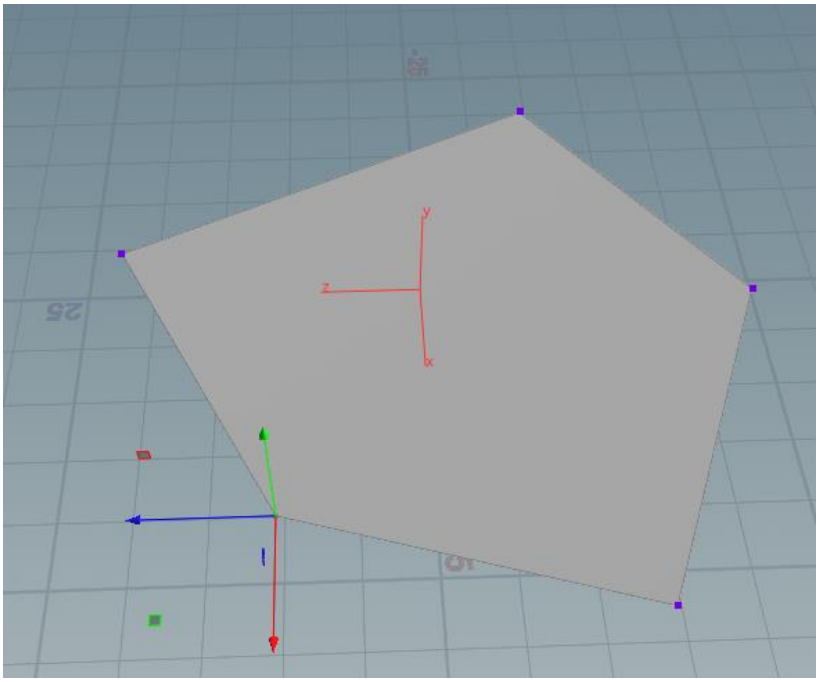


Image 8: Curve polygon generation

Once the geometry is already set-up, each edge is subdivided in order to have a point each wall width unit (2 per default).

After the subdivision is set, the borders are extruded inwards, then everything but the interior edge is deleted, and now is extruded outwards the same amount. This process helps fix overlaps caused by the first extrude.

Then, the whole polygon is divided in primitives' groups 3 by 3, which each group will end up being an individual building.

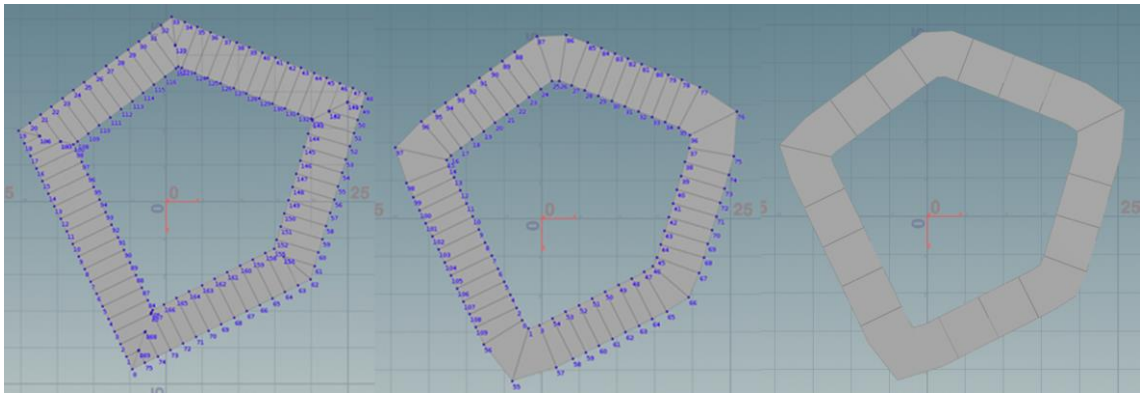


Image 9: Apartment buildings distribute steps

Now, each primitive (also known as building) is processed individually in the next steps and all are merged together at the end.

3.1.2 WALLS, WINDOWS AND DOOR

In the first hand, the walls have to be created. This building has two types of walls, the ground floor (which is taller) and the other floors above the ground floor.

The base is extruded to create the first floor (4 units by default), then the other geometry is translated by the amount of extrusion from this floor, placing the second floor just above it.

Then, the second floor is extruded (3 units by default), and after it has been extruded, is copied the desired number of floors and translated upwards by the other floors' height.

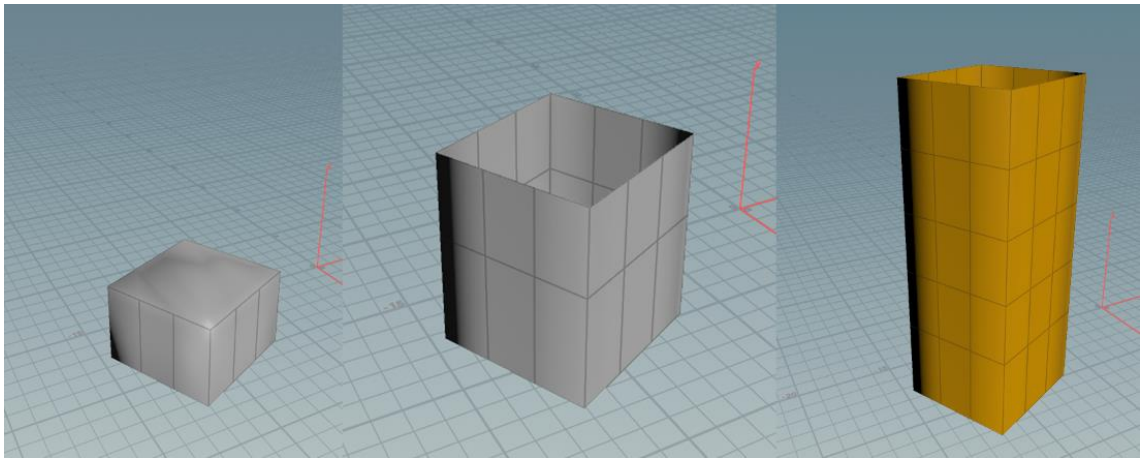


Image 10: Apartment buildings floor build-up

This is the base wall for the building. Now, in order to get the position to add the windows, using the previous outer and inner edge from the main polygon, selects the points in contact and delete everything else. This generates the front and back points for the windows (Won't be windows on building's side wall, as it can be blocked by another building taller than the current one).

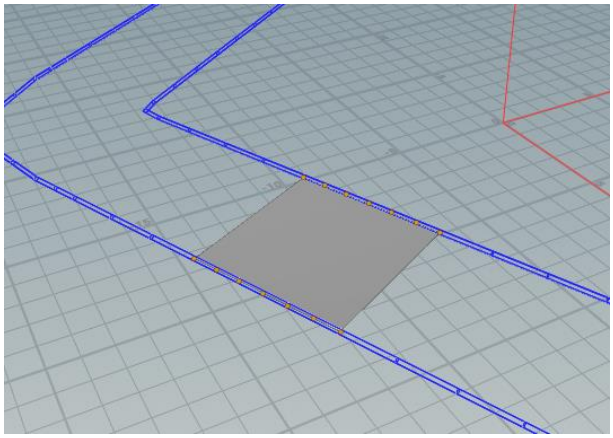


Image 11: Apartment select exterior points

These points are subdivided by 2 and deleted the previous ones, so they are now in the middle. Then, using the main walls, the normal attribute is transferred to each point, setting it to point perpendicularly from the wall.

Now, each group of points is copied and translated to its corresponding floor, fitting always in the middle height of each floor, so the windows are always centred.

The window mesh is a combination of a cube and half tube, merged together. As per the frame, it's just playing with the size and using Booleans to subtract the middle geometry.

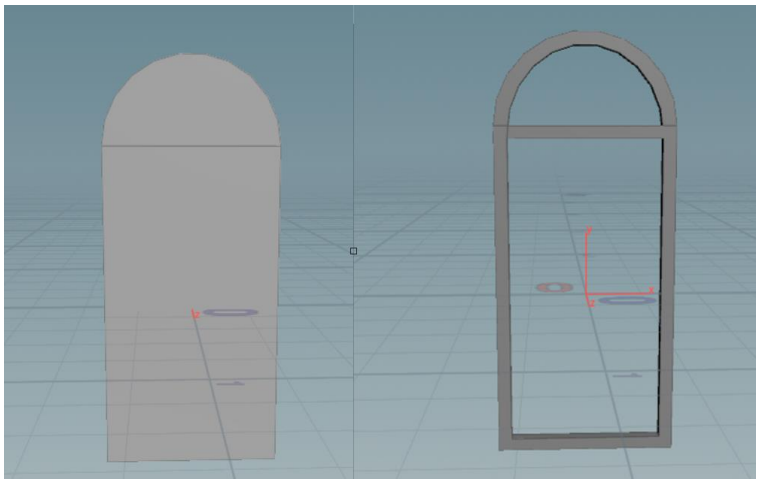


Image 12 Apartment windows meshes

To create the windows in the mesh, first the left mesh is used as bounding volume to create a whole in the wall with a Boolean. Then, the frame is translated to the whole. For the glass, it's just the resulting plane from the Boolean subtraction.

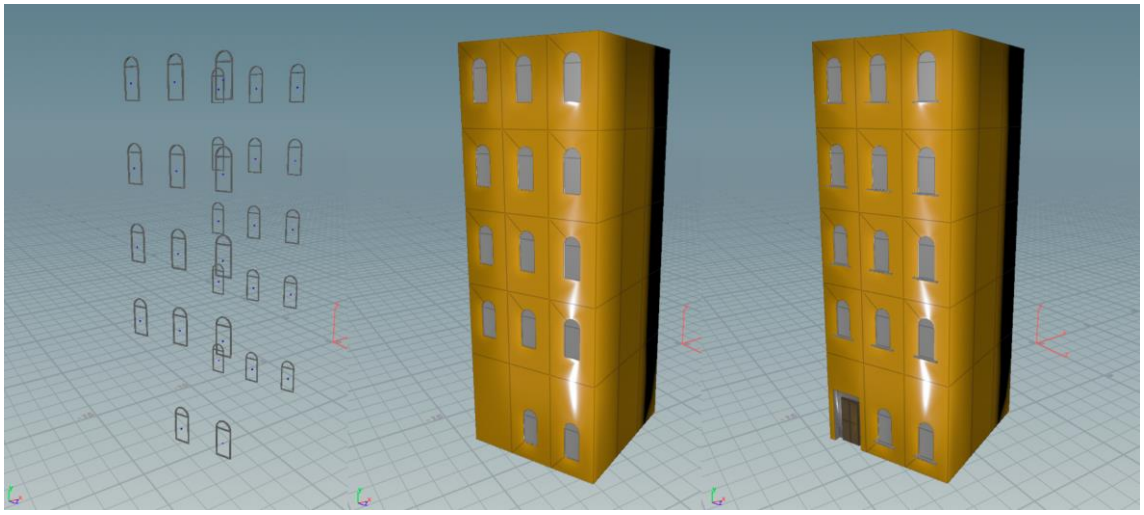


Image 13: Apartment buildings windows build-up

For the door, it just uses one of the ground floor window points and follows the same procedure as the window. It just has a bounding volume (bigger than the door size) that creates a hole in the building and then the door is placed in that hole, fitting perfectly.

Each window also has a ledge. This ledge is placed in the same point as everything used for the windows, then it's translated by half the size of the window height plus half its height, fitting pixel-perfect in the lower part of the window.

3.1.3 CORNER BRICKS

For the corner bricks, it uses the points created when each primitive was created in the beginning, before they were subdivided. This way, these points are just placed in the corners of each building.

After that, it's as easy as copy the blocks to the corners, duplicate them and place each one in a different floor.

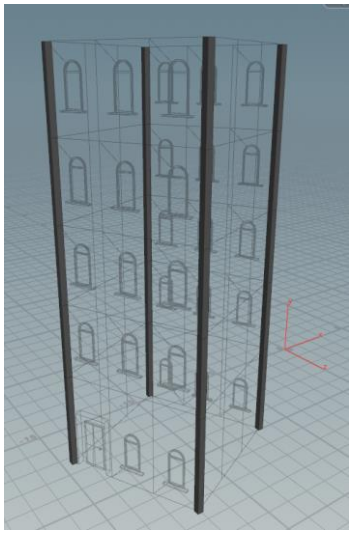


Image 14: Apartment buildings corner blocks

3.1.4 ROOF

There are two types of roofs for the apartment building, and each one is assigned randomly in each iteration.

The first type uses the base geometry placed on the top of the building.

The second one, uses the same geometry as a base. Then, this geometry is extruded inwards, deleted everything except for the inner edges and extruded outwards the same amount. After this process, the inner edges are lifted up and the whole in the middle is filled.

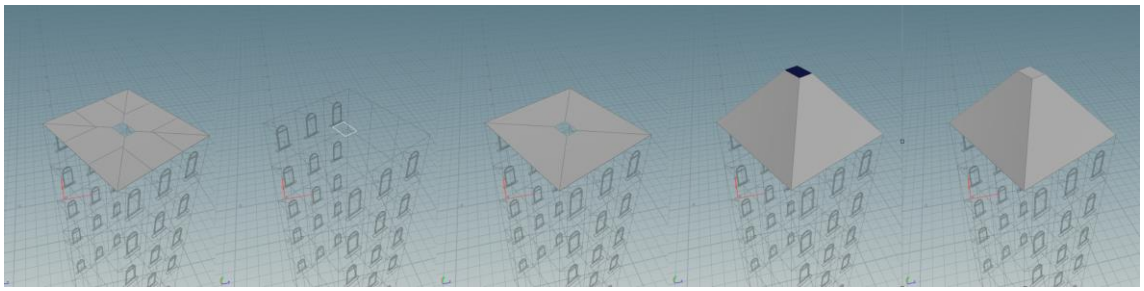


Image 15: Apartment buildings roof buil-up steps

Moreover, this base geometry is used to generate a border. After extruding it, results in a nice roof ledge.

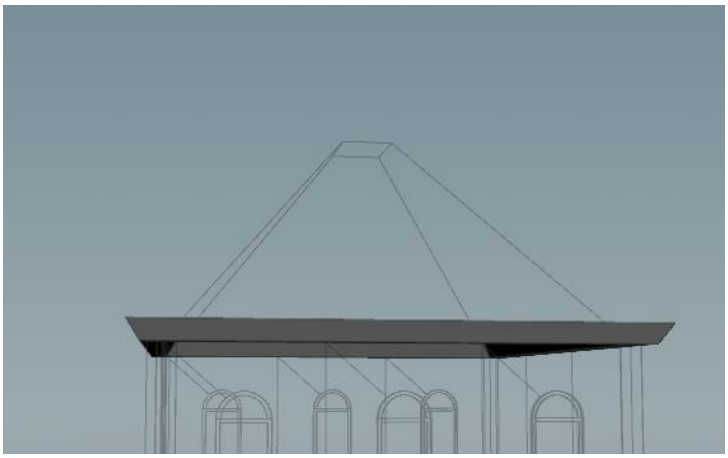


Image 16: Apartment roof ledge

3.2 SKYSCRAPER BUILDING GENERATION

In this point, the generation of the skyscraper type of building will be discussed.

3.2.1 BUILD-UP BASE STRUCTURE

For the generation of this type of building, it has almost the same properties as the apartment building.

Even though there is no different approach for generating the building from a procedural generation or a given shape.

Once the asset has received the input shape (procedural or not), in order to maintain the point density, the resample node is used. This node adds points in each edge based on the length given ("Wall Width"), so this is essential to maintain the same size in each wall independently of the building type.

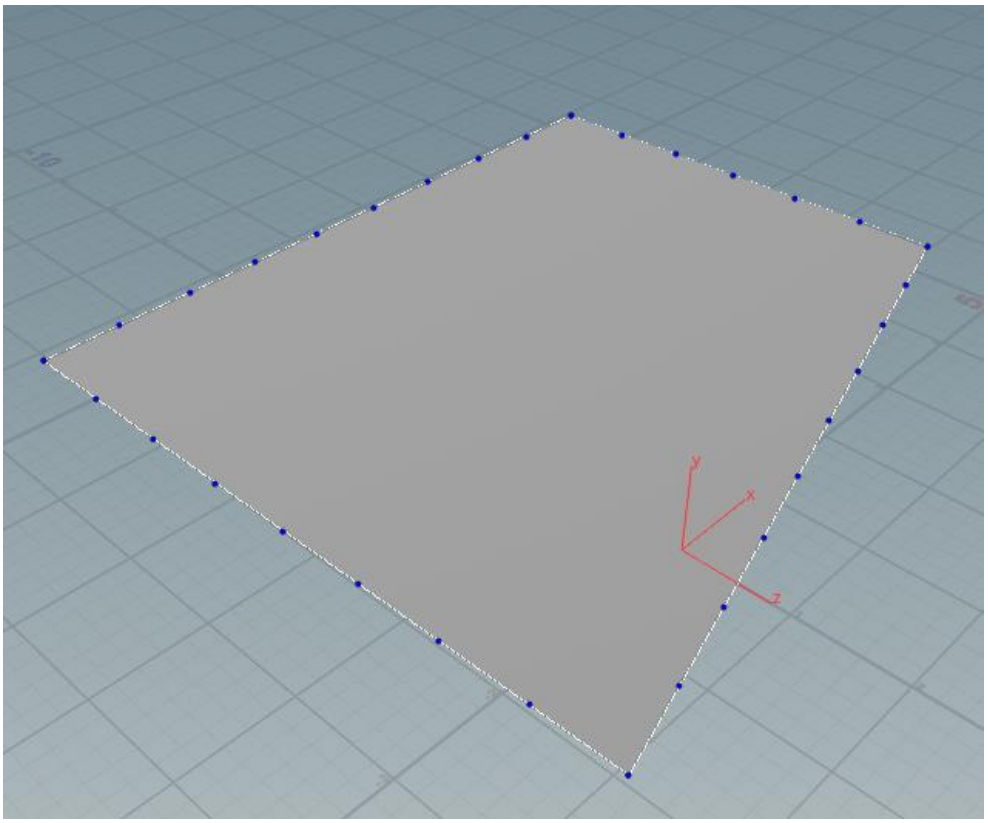


Image 17: Resampled geometry

3.2.2 WALLS AND WINDOWS

Once the base geometry has been resampled to have the appropriate number of points, the geometry is extruded in the positive Y axis, by the wall height, and because of is not needed anymore, the inside geometry is removed.

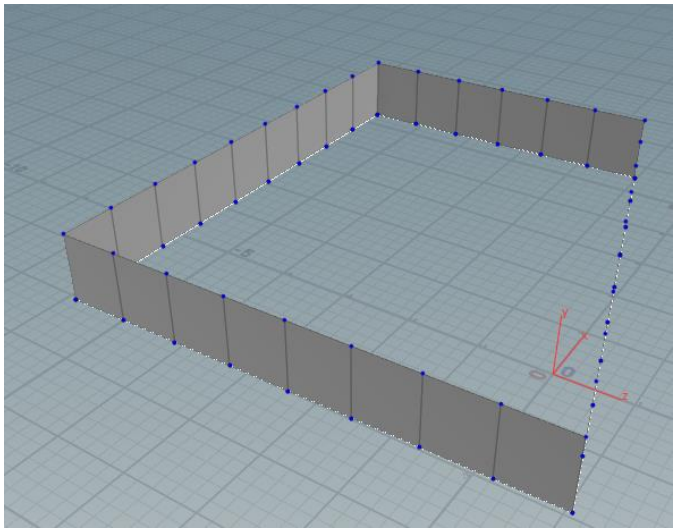


Image 18: Extruded wall geometry

Now, to select the door, the same procedure of the apartment building is used, but instead of selecting a point, a primitive is selected and separated from the rest, as it'll be the door.

Then, the geometry is duplicated the number of floors and translated the "Wall Height" value in each iteration. After that, each primitive is inset inwards and adding the inner primitive into a group. Then, this group is extruded inwards by a small amount, creating the windows.

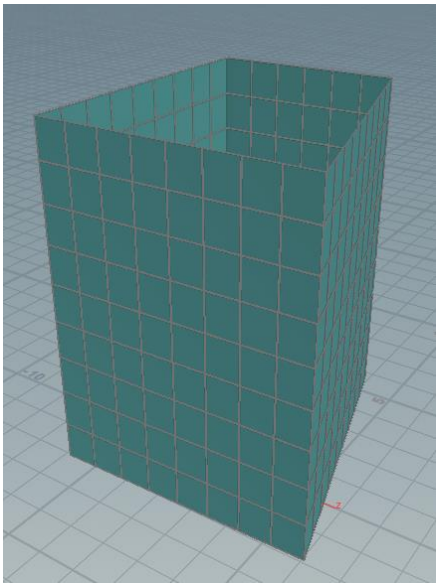


Image 19: Final wall setup result

3.2.3 ROOF

The generation of the roof is pretty simple, it just uses the input geometry but translated in the positive Y the amount of floors * the wall height.

For adding a little more detail, the geometry is inset inwards and that part is extruded upwards, creating a little border.

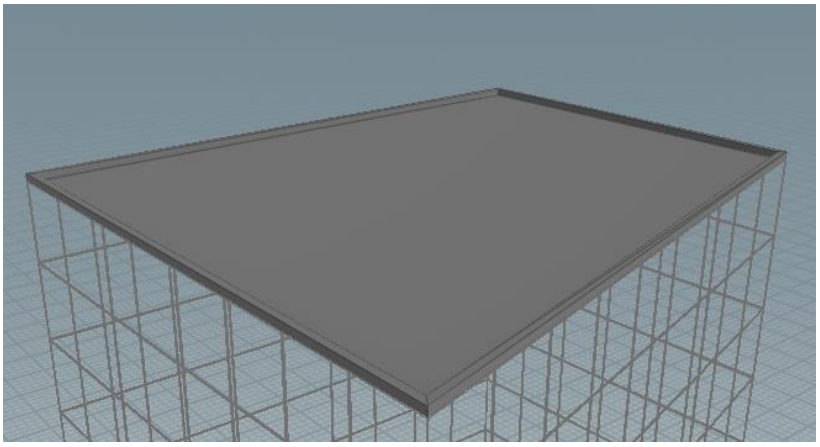


Image 20: Roof geometry

3.3 PROCEDURAL CITY GENERATION

In this point, the different approaches for procedurally generating the city will be discussed.

In order to generate the city, there are two types of generation. One starts with a black and white image and extracts the building information (white). The other one, just generates the building placing in a plane by slicing it using some mathematical distributions such as Voronoi and Manhattan.

3.3.1 READING IMAGES

In this implementation, the asset receives a simple black and white image and compute all the white information in order to generate its edges.

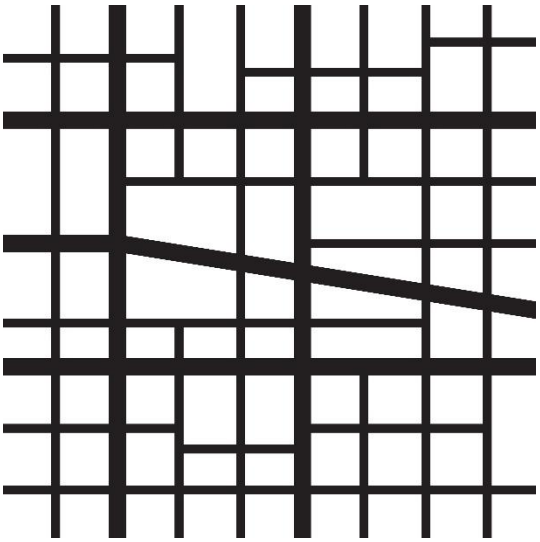


Image 21: Example image input for the city generation

After being processed, it looks this way in the editor.

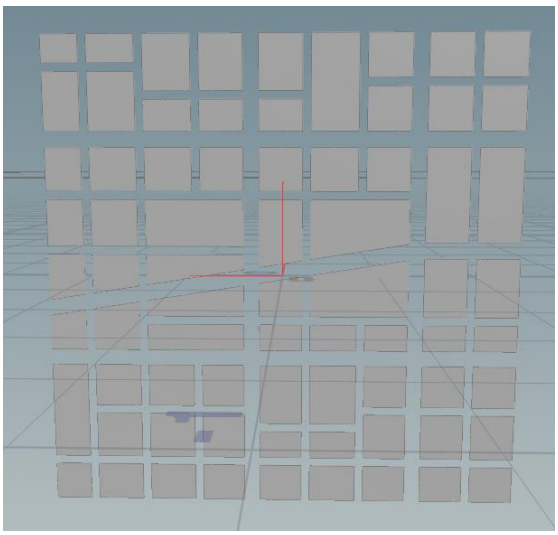


Image 22: Geometry after processing the image in Houdini

Then it's translated and scaled, in order to get the correct size for the buildings (As they use the world length for making the wall divisions).

After that, a resample is done in order to work with a less density mesh, as it's faster for the future calculations.

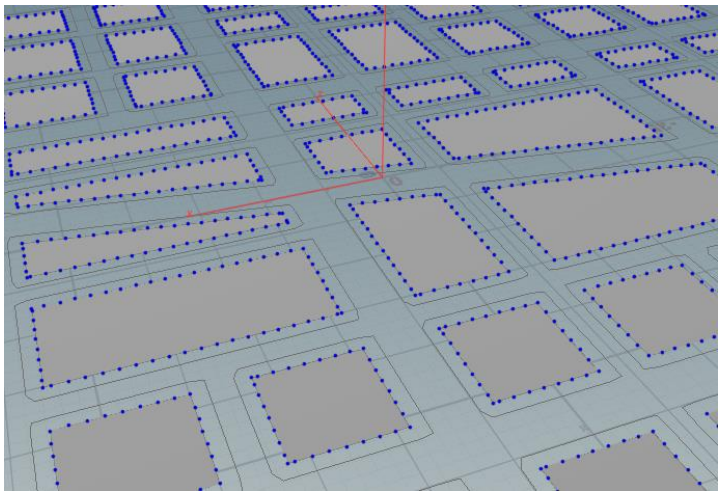


Image 23: Base city geometry

3.3.2 MATHEMATICAL DISTRIBUTIONS

There are some types of distributions that can fit in order to build a city layout.

The first one implemented is the Voronoi distribution [2], which uses points on a plane in order to expand from each one and when the limits of two expansions meets, it generates an edge.

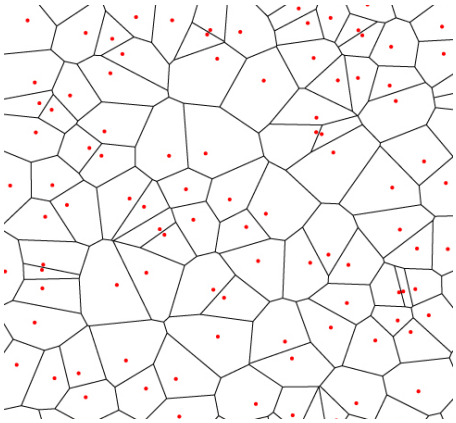


Image 24: Voronoi distribution

As Houdini already have a Voronoi Fracture node implemented, the procedure to make it work is to create a plane with (1000x1000) unit size, and then scatter some points into it. Then these points are creating the Voronoi polygons.

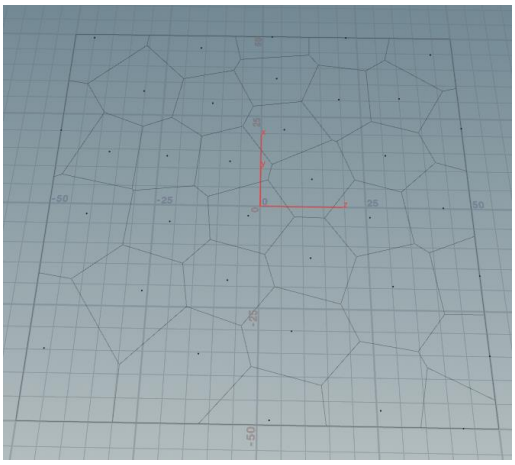


Image 25: Voronoi distribution in Houdini

Then, each primitive is separated from the rest by extruding inwards an amount corresponding to the half of the street size.

Then, feeding this geometry to the building generation tool, this is the result.

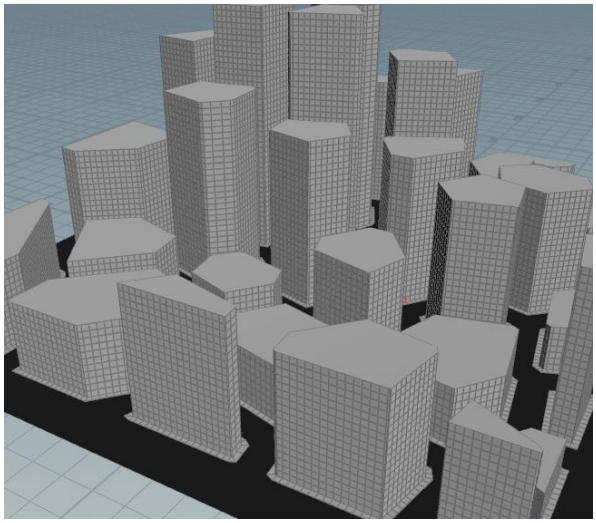


Image 26: Voronoi generated city

Moreover, other types of distribution have been considered to find the one which adapts the most to the city layout. For instance, a Manhattan distribution is one that have been implemented but have ended being discarded as it wasn't representing any actual architectural shape for a building.

To implement this distribution, a custom script is made from the result of the Voronoi distribution.

All the points are checked to get the ones that aren't endpoints and they're slightly modified.

The points are pushed inwards or outwards depending on the distance from the current point to its endpoint. To do this, this custom VEX script is needed.

```
//Get the neighbouring points
int n[] = neighbours(0,@ptnum);

//Check point is a center, not an end
if (len(n)==2){
    //Get connected end point
    int i = neighbourcount(0,n[0])==3?n[0]:n[1];
    vector pt = point(0,'P',i);

    //Find difference vector between our point and the end point
    vector d = @P-pt;

    vector pos;
    if(abs(d.x)>=abs(d.z)){
        pos = set(@P.x,@P.y,pt.z);
    }else{
        pos = set(pt.x,@P.y,@P.z);
    }

    @P = pos;
}
```

Image 27: Mahnattan distribution VEX expression

Then, once applied all the other modifiers (Extrusion, attributes, etc) the city is sent to the tool as input parameter for having the base geometry of the city, and this is the result.

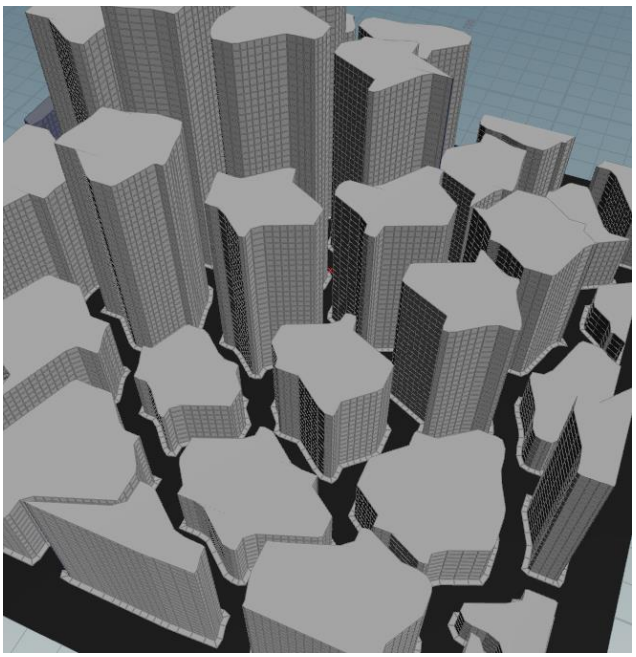


Image 28: Manhattan distribution city

3.3.3 FINAL GENERATION APPROXIMATION

As the city layout is based in New York, looking the city from above shows how the layout can be replicated, using a mix of the previous techniques explained.

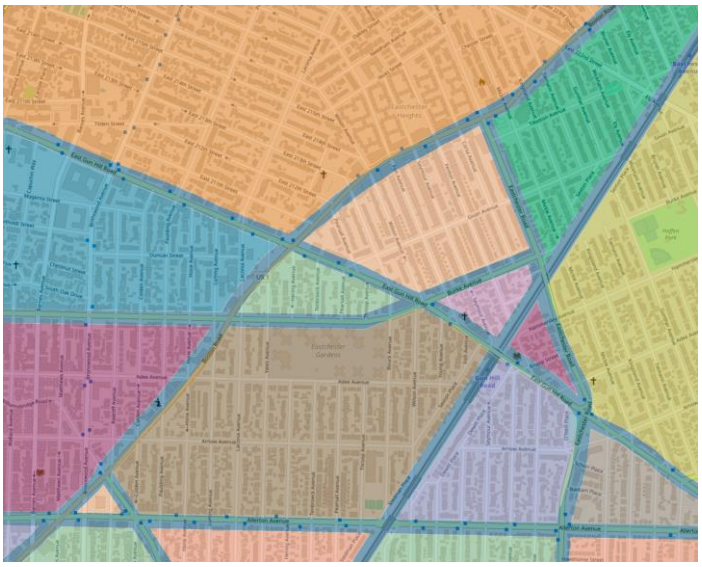


Image 29: New York map divided

The map shows the principal roads in colour orange or yellow, these roads are similar to the Voronoi distribution. Then, each zone between the main roads, has more small roads following a grid pattern. Therefore, combining a Voronoi and a grid layout, gives some similar results.

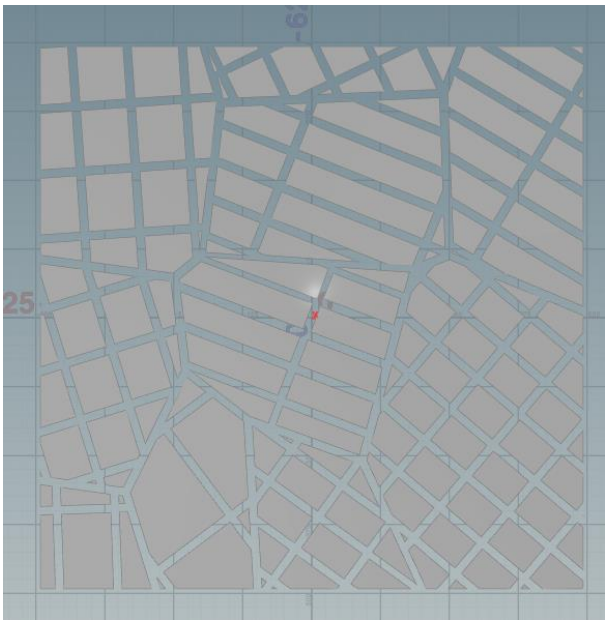


Image 30: Combined Voronoi and grid distribution

3.3.4 DISTRIBUTING THE BUILDINGS

It is common that in the cities the type of building is predominated by the zone itself. Hence the distribution of them is focused on recreate that pattern.

To approximate to this representation, a single polygon from the generated layout is picked and set as the type of building 1. Then, it is extended to its nearby polygons within a radius of influence.

To add more variety some of the other polys (that haven't any type of building yet) are added to them by a random percentage.

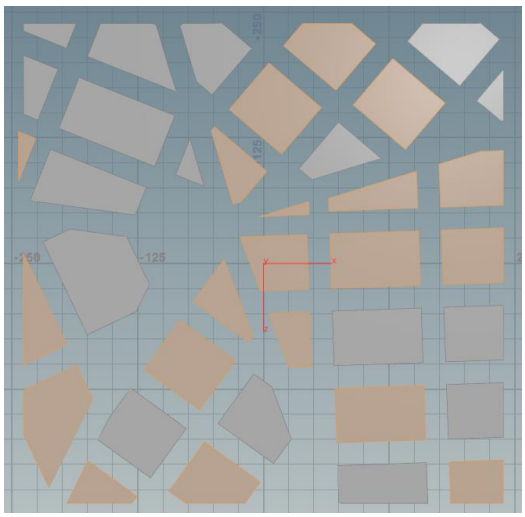


Image 31: Building zones distribution

3.3.5 DISTRIBUTING THE STREETLIGHTS

The base geometry is used to distribute the streetlights. This distribution relies on calculating the border edges of the polygons, select a percentage of points and place a streetlight at each point.

The geometry is extruded inwards, so the point remains inside the pavement zone. For this extrusion is also considered some of the traffic regulations[3] that indicates where does the streetlights go in the sidewalk.

Then the priority is to calculate the normal attribute of these points. It must be aligned perpendicularly to the pavement border, so each light will be pointing towards the road.

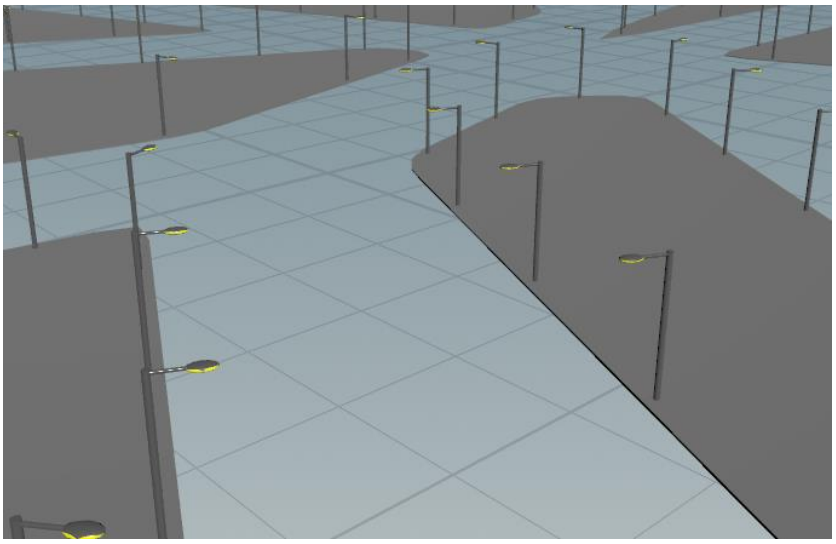


Image 32: Streetlights distribution in Houdini

The streetlight itself is made of two tubes and a deformed sphere (which its low part is set to have an emissive material), and every part is related to each other so when something is transformed, everything goes along.

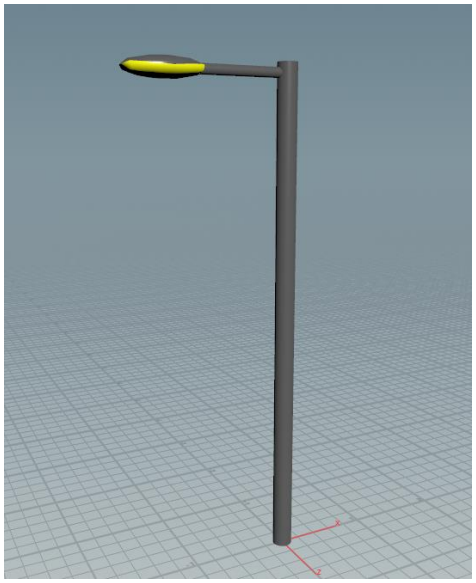


Image 33: Streetlight mesh

3.3.6 GENERATING THE SIDEWALK

When generating the sidewalk, the traffic regulations[3] are really useful to establish the maximum and minimum size of the sidewalk (So the result is more similar to reality).

To generate this, is as easy as doing a loop over each primitive (each building base polygon) and inset inwards the desired amount between the minimum and maximum width.

Then, the outer part of the resulting geometry is extruded outwards to create the sidewalk border stones. These stones are irregular in real life, there aren't two with the same exact height. Therefore, using a slight random value to add some variability results in a very plausible geometry.

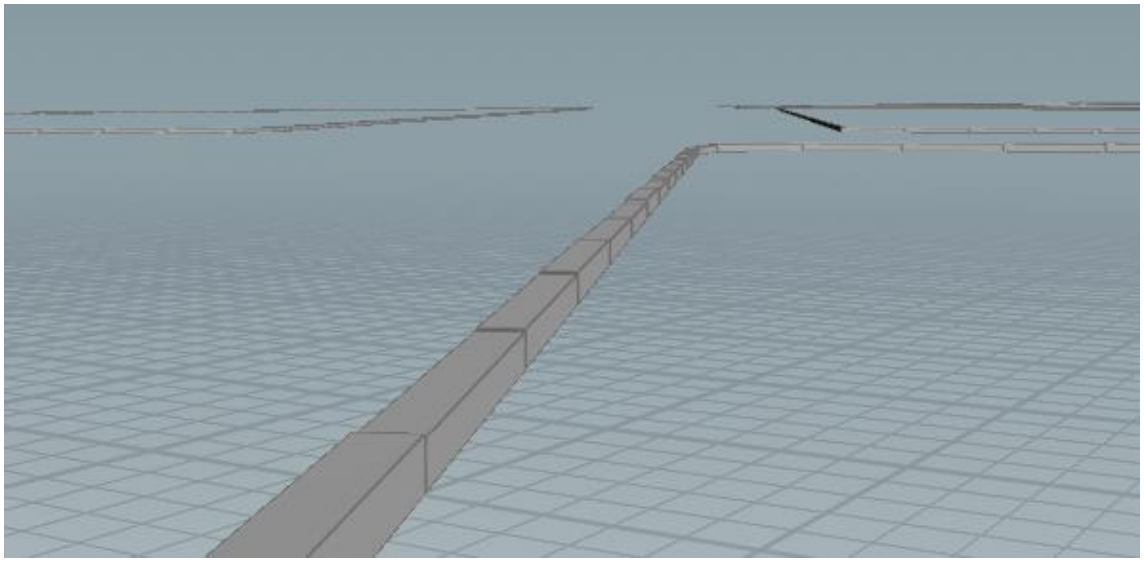


Image 34: Sidewalk border height variation

3.3.7 DISTRIBUTING THE MANHOLES

Every city has manholes in the streets, so it's something it has to be when a city is built.

For the distribution, cities follow all the underground distribution, but this pattern is not even close to be visible by people from streets, so it results in a more random distribution from that perspective.

In order to distribute the manholes the approach was to take into consideration the density of the distribution in each sub-part of the city. This means, that the city is subdivided into a smaller grid cells, and for each cell the points are scattered. This grants a more equality in terms of distributing the manholes.

Once the points are scattered, the ones that remained inside the buildings (they'll be invisible at the end) are removed due to optimization. Then, the ones that its distance from the sidewalk or to other manholes are less than its current radius, are removed too.

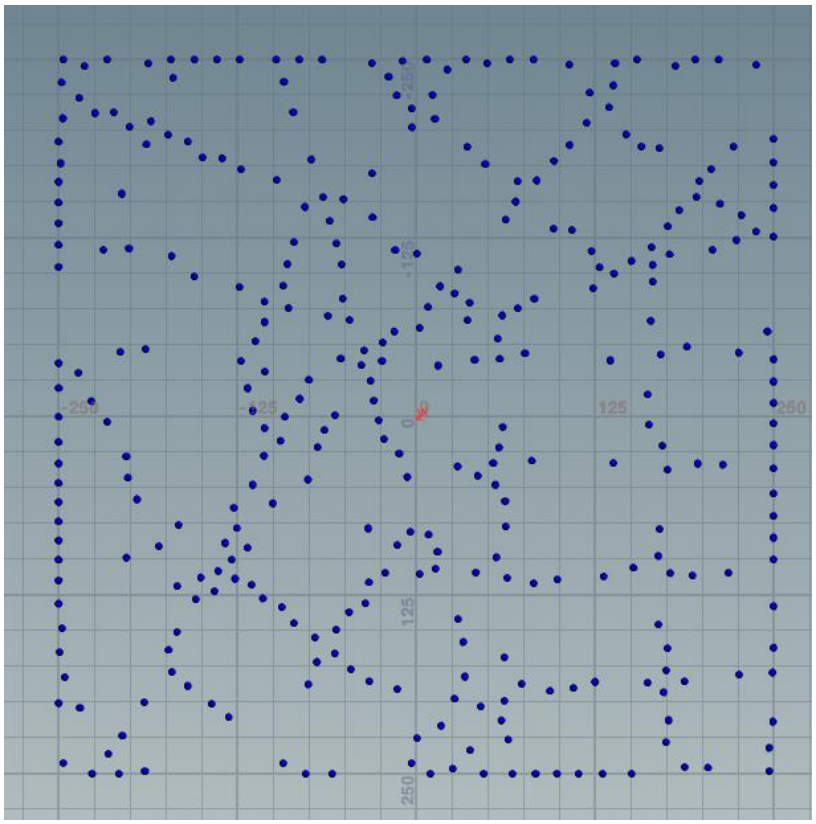


Image 35: Manhole distribution

3.4 TRAFFIC STREET LINES

Every road has traffic lines to guide the vehicles within each lane. There are some ways to achieve this: By adding decals, geometry, etc.

The followed approach is to add some small geometry, as it works better with Houdini than decals or projections. The procedure is to, given the intersected edges from the layout creation, generate curves to place the geometry along them.

One of the concerns considered in this approximation is that in the intersections, these lines must be discontinued to avoid confusion of lanes.

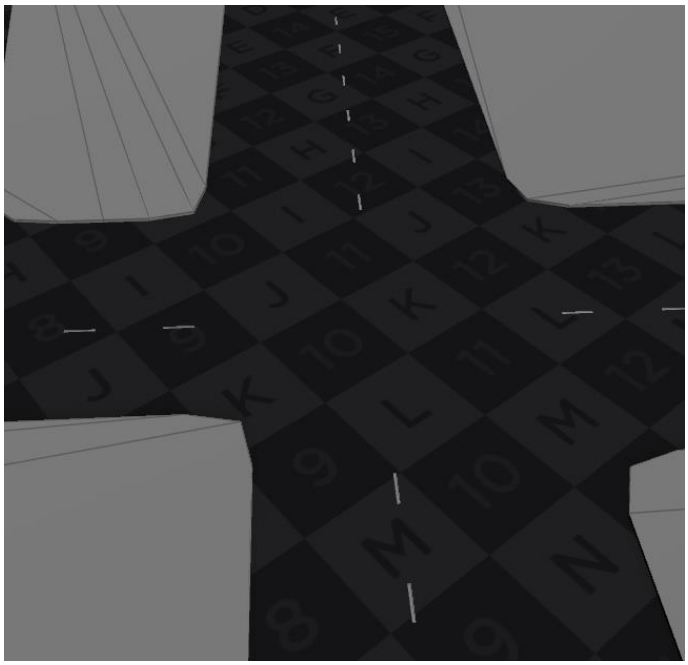


Image 36: Street lines intersection

3.5 PARKS

Every single city needs green zones. To achieve this, the top 2 or 3 biggest areas generated from the city layout, are chosen to be a green zone.

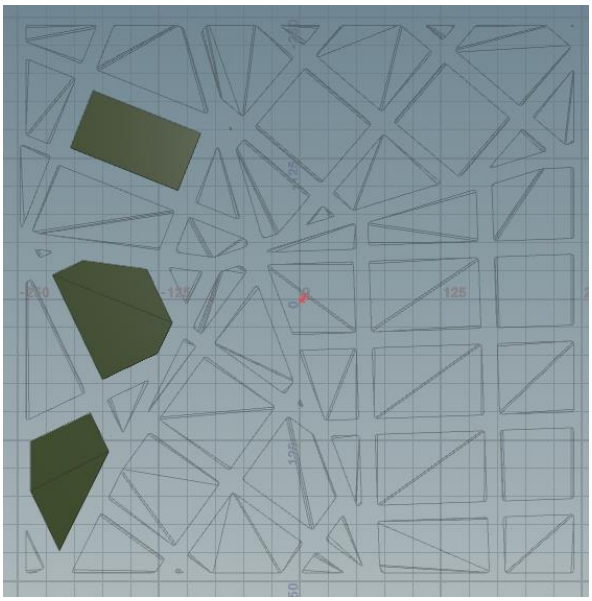


Image 37: Park selected base geo

The objective in this part is to add some vegetation, terrain variation, some small lakes, etc. But since there's not possible to implement LODs through the asset, there's no point of adding them to the asset as it won't be possible to render it in real-time. Although it was implemented anyways using a terrain with some noise applied, splines to make the paths and then trees scattered along the surface.

Image 38: Park model with trees

3.6 UV OPTIMIZATION FOR MATERIALS

In this point everything related to the materials and the distribution of the multiple UV channels will be discussed.

In order to optimize the texture streaming in the final scene, some considerations have been taken.

The problem is that these assets are large, so a simple UV channel per model or part of the model won't be enough and it will require a super high-resolution texture in order to have enough quality.

To minimize this problem, the solution is influenced by the texture workflow currently used at DICE[1]. In which in each asset have two UVs layouts, one that have all the similar geometry stacked and the other that have everything unwrapped without any overlapping.

In order to understand their workflow, this is an example.

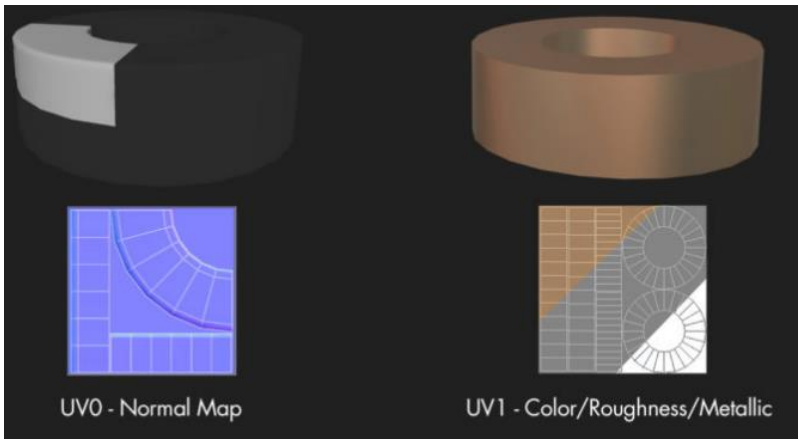


Image 39: UV workflow example from DICE

In the UV0 they use a stacked geometry so they can lower the normal resolution (This will be my UV1 channel for textures) and in the UV0 they have all the geometry unfolded in order to add unique detail (This will be my UV0 channel for Lightmap).

3.6.1 LIGHTMAP UVs

The lightmap UVs are placed in the UV channel 0 and it's the default channel used by almost any game engine to bake the shadows in the static meshes.

Therefore, as the shadow will be different in each part of the model, this layout must be overlapping free, and it won't require a super high-res texture as shadow maps usually uses small resolutions (it's enough).

In this case, if we tried to use this layout to add a material, the result will have lower-res than it should be. Therefore, for fixing this problem the detail UVs layout is made.

3.6.2 DETAIL UVs

This UV layout uses the UV channel 1 which is not used by default by the engine.

The advantage of this layout is that all the geometry is stacked in similar stacks, allowing each piece to have more space inside the UV cords. Therefore, if each primitive has more space, this means that will have more resolution when a material is applied to the model.

The problem is that if the geometry is stacked together, there will be problems with the texture as the tiling would be too much noticeable.

To fix this, a custom shader is made in order to use a triplanar distribution of the texture and to use the correct UV channel.

The difference between each UVs packing is quite visible. For a simple cube the Lightmap UVs (left) has around 55% coverage, while detail UVs has around 80% coverage.

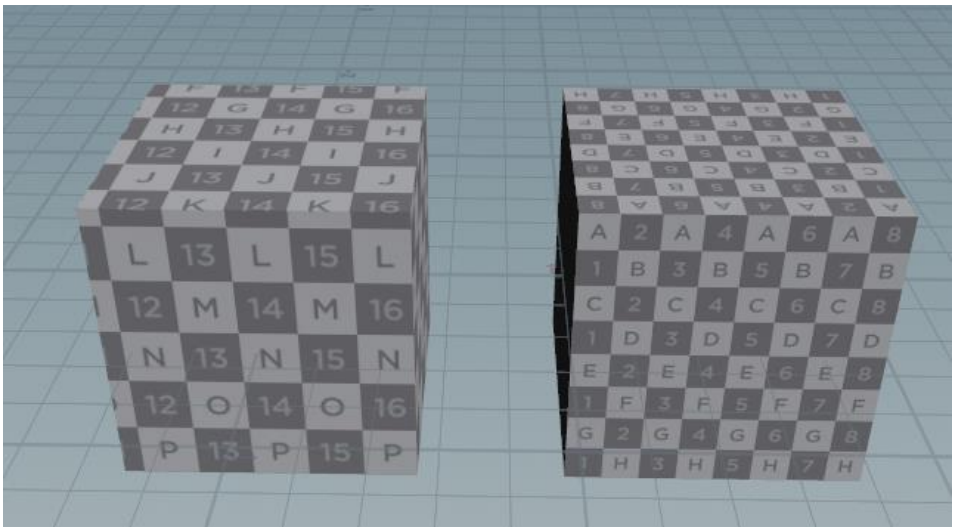


Image 40: Non-overlap uvs Vs stacked uvs

As can be seen, the texel density has increased, allowing to use lower resolution textures and grant a high-quality asset.

The counterpart of this technique is that, nowadays Houdini just don't support multiple UVs channels through the asset. Therefore, the implementation is not usable. The final implementation will only use the detail UVs, as geometry can still be static except for baked lighting and it gives a lot more realism.

3.6.3 MATERIALS

For the materials, the triplanar shader is created using the Unity PBR Shader graph. It's just a standard PBR shader with Albedo, Normal, Roughness and Metallic inputs using triplanar projection in the UV1.

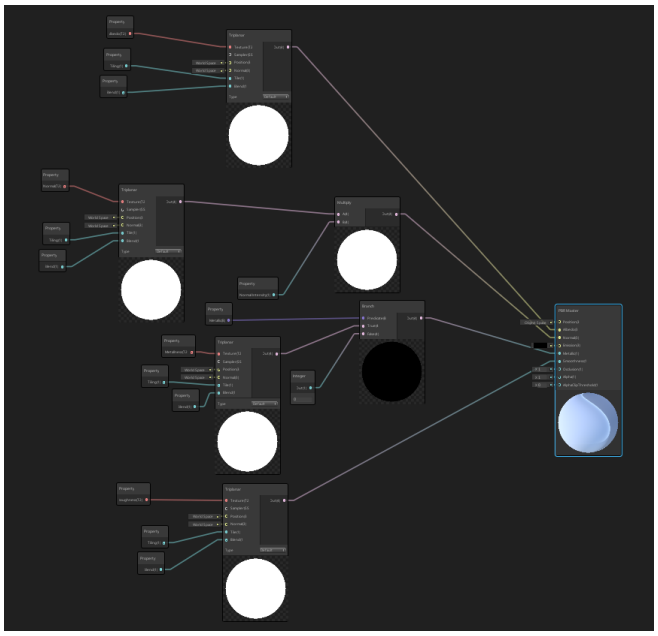


Image 41: Shader graph Triplanar PBR shader using UV1 channel

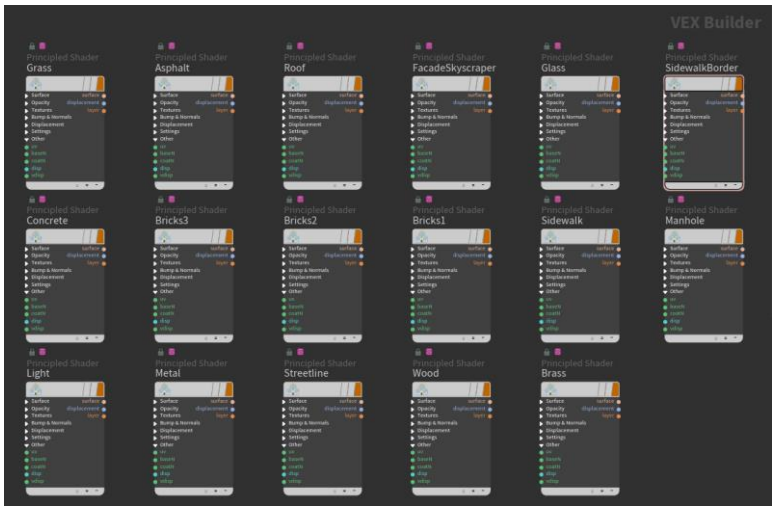


Image 42: Houdini material setup

The other problem with working in materials in Unity is that Houdini don't support Unity materials as inputs, the way it's made inside the program is by giving a string with the relative path of the material inside the unity project.

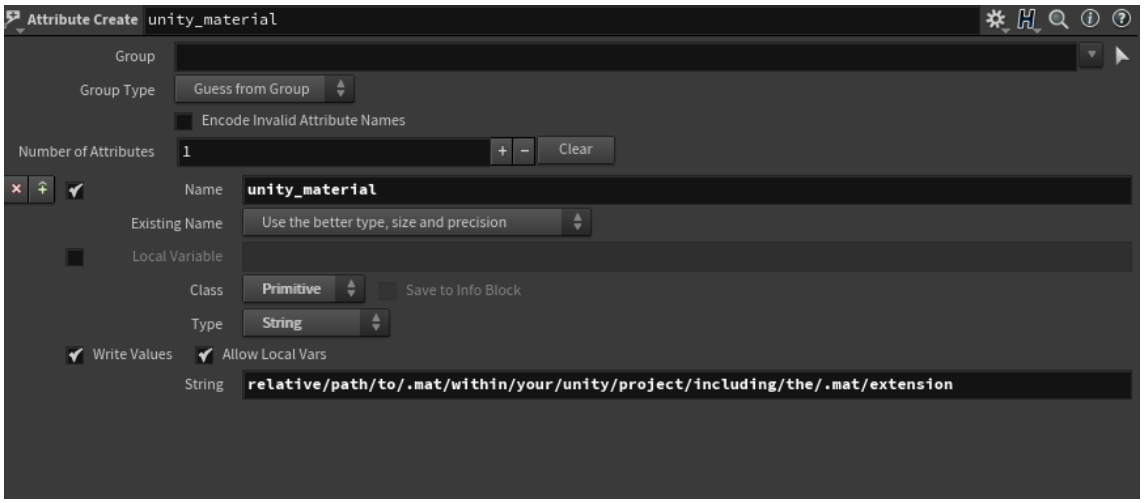


Image 43: Unity material attribute in Houdini

This way of adding materials is completely inefficient, so in order to fix this a custom C# script is made. The script receives materials as inputs, gets their relative path and sends them to the Houdini HDA (Asset controller). In order to send the paths more efficiently a custom editor is implemented with a button to launch it directly from the editor.

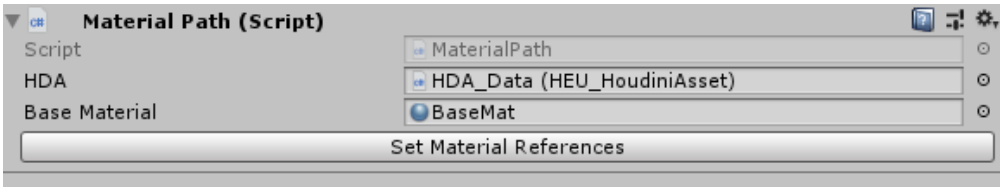


Image 44: Unity Custom editor button

4. RESULTS

In this chapter, the results of the development process will be shown and discussed.

After finishing the development of the project, the final result has been achieved. So, for each main objective, a result will be shown.

These results consist in procedural apartment, procedural skyscraper and procedural city.

They can be found in the Appendix B.

4.1 PROCEDURAL APARTMENT

The apartment is the first objective started and reached in the project, and it's the one that has the most time involved in it.

As it was said in the objectives, one of them is to implement this tool into Unity 3D. Therefore, showing the result of the tool directly used in unity is the most appropriate way.

Image 45: Example generation of apartment buildings

So, as it can be seen, lots of variations can be generated.

This is how the interface looks like inside Unity

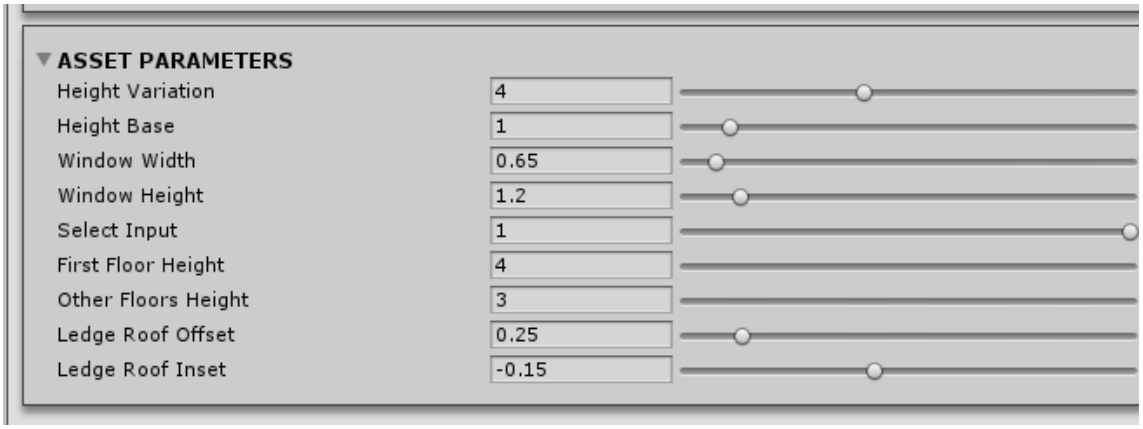


Image 46: Procedural apartment asset menu in unity

There are the nodes that drives all the procedures and logic of the apartment generator asset.

The base layout looks like this.

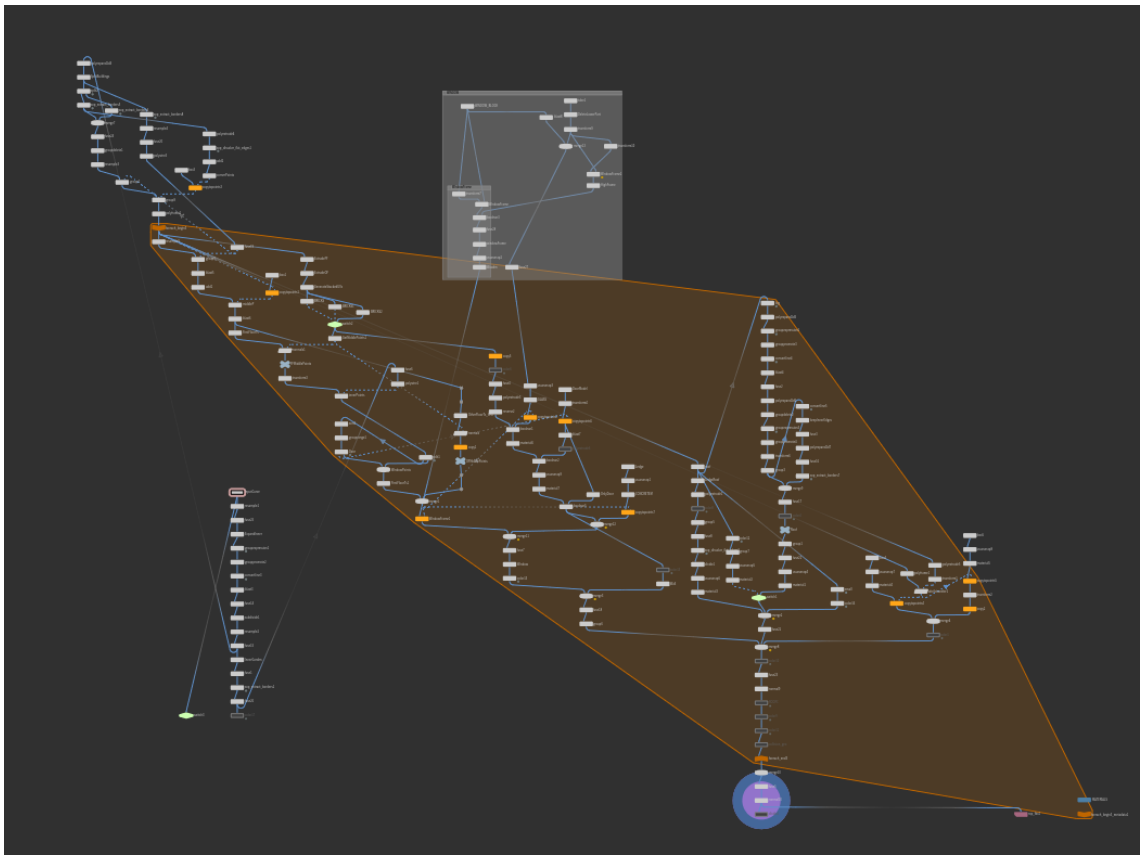


Image 47: Node distribution procedural apartment building

For a closer view, the node layout is shown in the appendix A2.

4.2 PROCEDURAL SKYSCRAPER

In this part the result of the skyscraper is shown along with all the nodes involved in its generation.

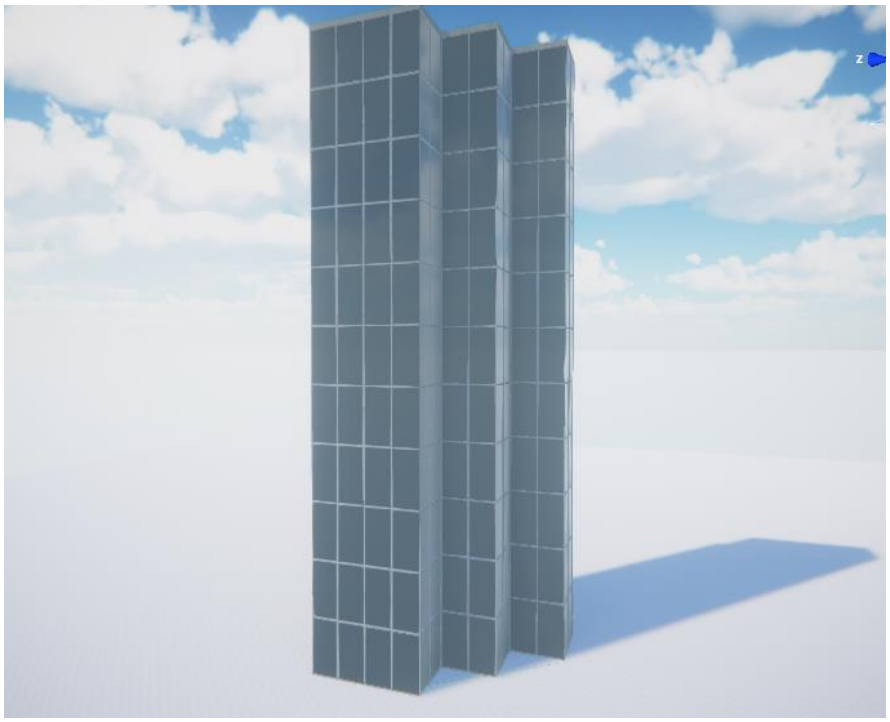


Image 48: Skyscraper building in Unity

This is how the interface looks like

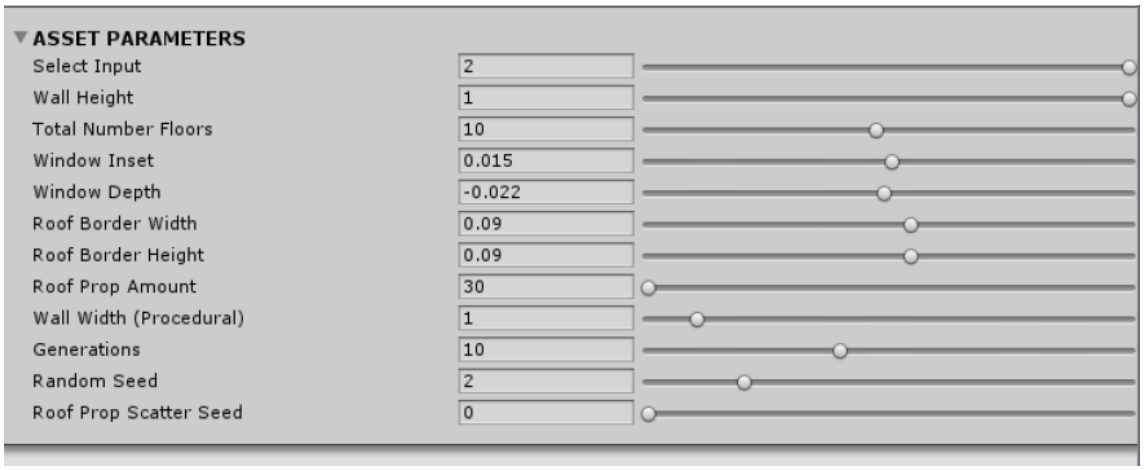


Image 49: Procedural skyscraper generator tool

More in deep node layout is shown in appendix A3

4.3 PROCEDURAL CITY

In this part, the final asset is shown. It creates a complete city from an input, from image or generated. This is the final result in Unity after applying the materials and everything.

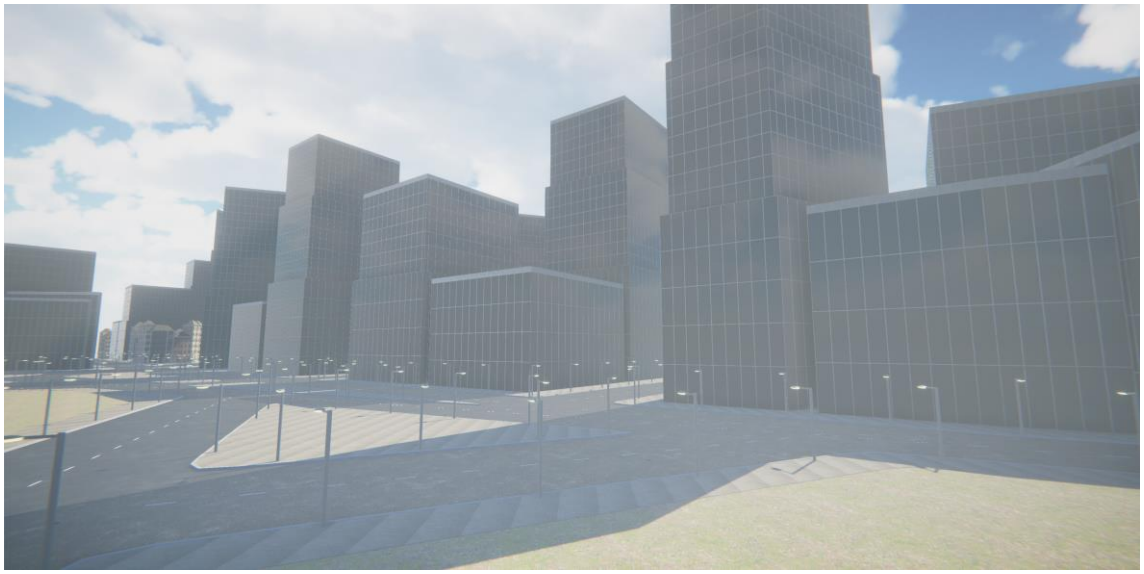


Image 50: Procedural city result Unity I

Image 50: Procedural city result Unity II

The resulting asset interface inside Unity integrates new parameters and some other parameters from the embedded assets.

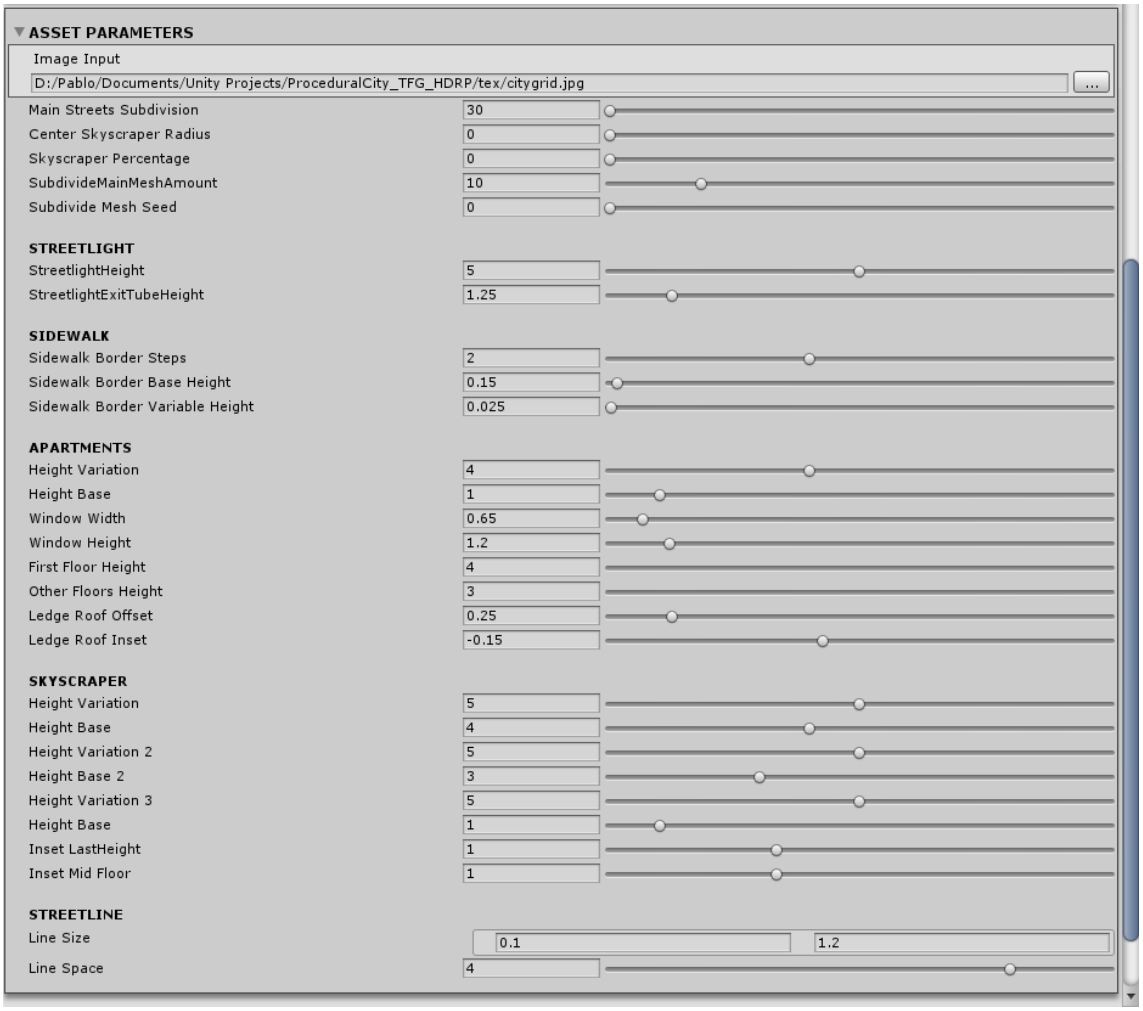


Image 52: Procedural city interface Unity

The generator nodes are structured in the following way.

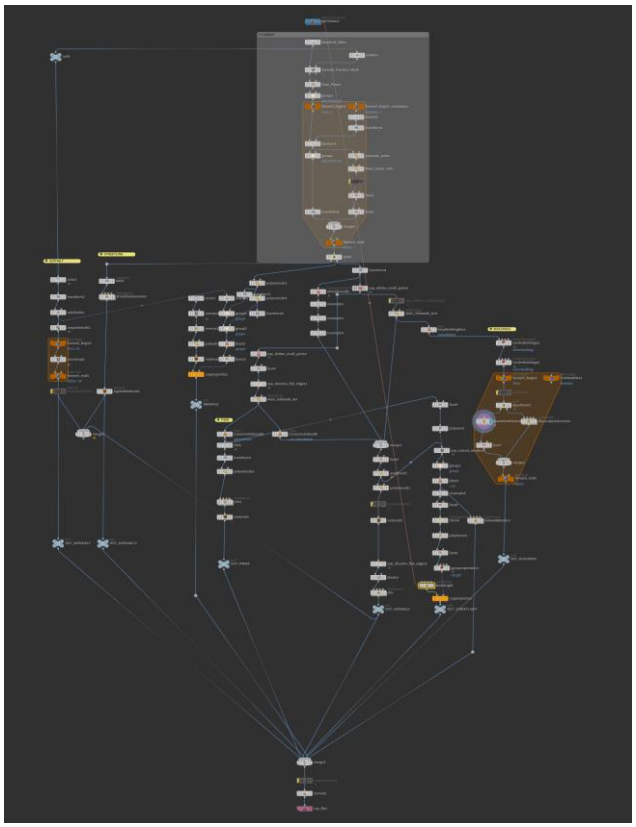


Image 53: Procedural city main nodes distribution

Each zone creates a model, for instance, the sidewalk, parks, street lines, etc.

The info each node is containing is placed in the appendix A4

4.1 FINAL SCHEDULE

The resulting Schedule may vary a little compared with the resulting one.

Task	Estimated duration (in hours)	Approximated real duration (in hours)
Documentation		
Technical proposal	8	10
Final report	50	40
Project video	4	8
Project defence preparation	12	10
Analysis		
Analysis and design document	24	20
Houdini introduction	20	30
References	6	4
Development		
Streets procedural generation	30	25
Buildings procedural generation	40	60
City procedural generation	60	75
Houdini plugin integration	12	20
Design		
Textures and shaders	8	4
Unity scene set-up	6	4
Testing		
Export result to Unity	20	16
Total	300	326

5. CONCLUSIONS & FUTURE WORK

5.1 CONCLUSIONS

Houdini Software is one of the most powerful and complex software used, it has a lot of power but it is essential to know the basics of it, as it even uses some different concepts (VOP, SOP, ROP, attributes, etc.) than any other 3D software.

The process of learning Houdini from scratch was tough and frustrating, as the lack of documentation and resources is something that Houdini has struggles nowadays.

But in the final period of the project, it was enjoyable as it was possible to transfer and apply almost all the ideas from the mind to the software.

The idea of building a procedural city from scratch is really complex, as cities are formed by lots of things and variety. So, in order to prioritize tasks, the main concept needed to be simplified, as by using Houdini was possible to generate lots of more complex things that were out of reach within the deadline.

In the end, all the expectations have been met. The objective of generating a city procedurally has been achieved and even surpassed as there is more than one way to generate the city.

The building generation works properly and has lots of customizable parameters, allowing end users to customize it as much as possible.

5.2 FUTURE WORK

Future plans have been thought for the asset. Lots of things were planned to be implemented but they ended up being out of budget.

The two most important tasks for future works are the LODs and Double UV channel, as they're not considered in the Houdini pipeline yet and is much needed to achieve a high optimization tool.

On the other hand, the plan is to have a very robust and customizable tool in order to add more variety to the generation of the cities and buildings. New props, parts and buildings will be added. For instance, some green zones such as trees, small lakes, playgrounds in the parks, store panels for the lower part of the buildings, train tracks, different roads heights (the train can go above the ground and cars will pass through a tunnel), a train station, traffic signs and traffic lights, rivers and bridges etc.

In the future plans it is also present the implementation of the Houdini PDG pipeline, as the current implementation don't take advantage of multi-processing and It just requires a high amount of time (As it have to calculate everything one by one). PDG is a procedural architecture designed to distribute tasks and manage dependencies to better scale and automate the content creation, allowing to process the information in multiple CPU cores. This process will make the tool even faster.

By using this integration, it is possible to generate the city in almost runtime. Therefore, generating an endless city is something that can be achievable by this update.

A APPENDIX

A.1. BIBLIOGRAPHY

[1] 80.lv - DICE texture workflow

https://80.lv/articles/001agt-dice-behind-the-art-of-battlefield-and-battlefront/

[2] Voronoi info

https://en.wikipedia.org/wiki/Voronoi_diagram

[3] Traffic regulation for street props distribution

https://www.buenosaires.gob.ar/desarrollourbano/manualdedisenourbano/paisaje-urbano-morfologia/resolucion-general-de-secciones-de-calles/calle-tipo

http://www.carreteros.org/normativa/trazado/31ic_2016/apartados/7.htm

A.2. PROCEDURAL APARTMENT

This appendix shows a little breakdown of the node structure inside Houdini used to generate the apartment.

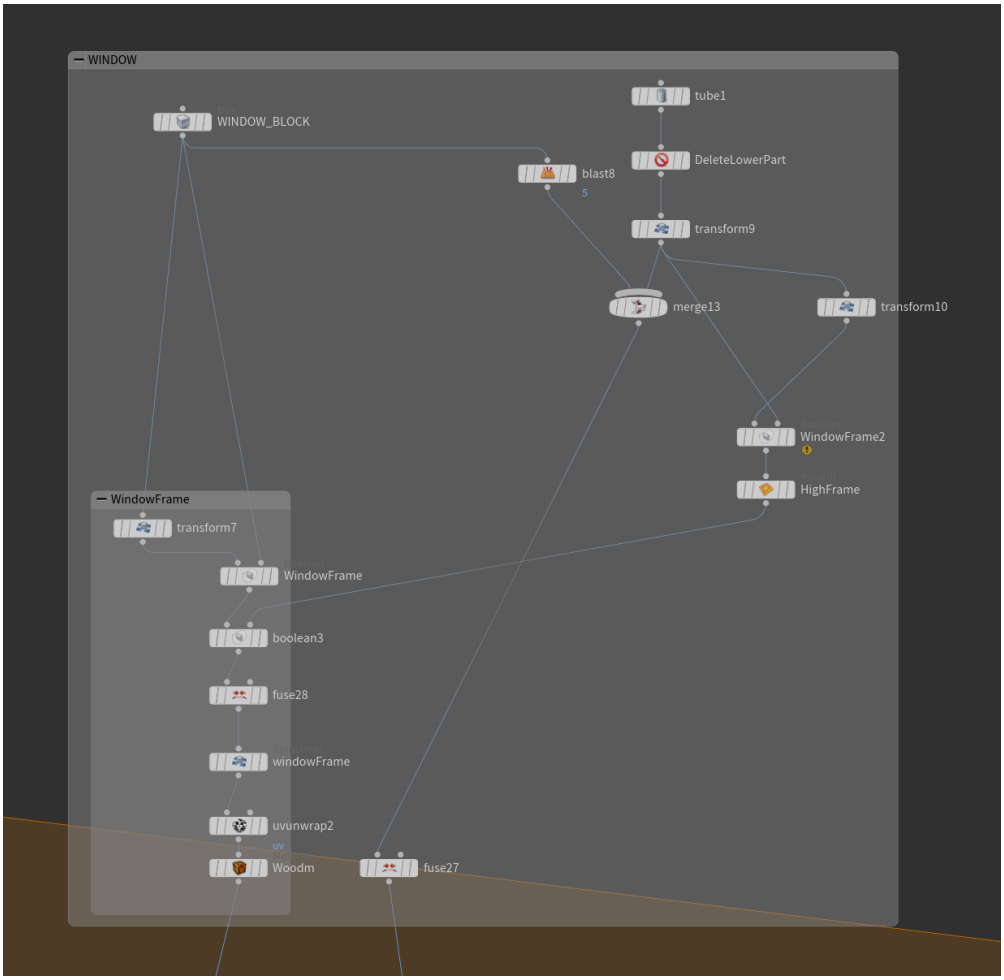


Image 54: Procedural apartment window nodes

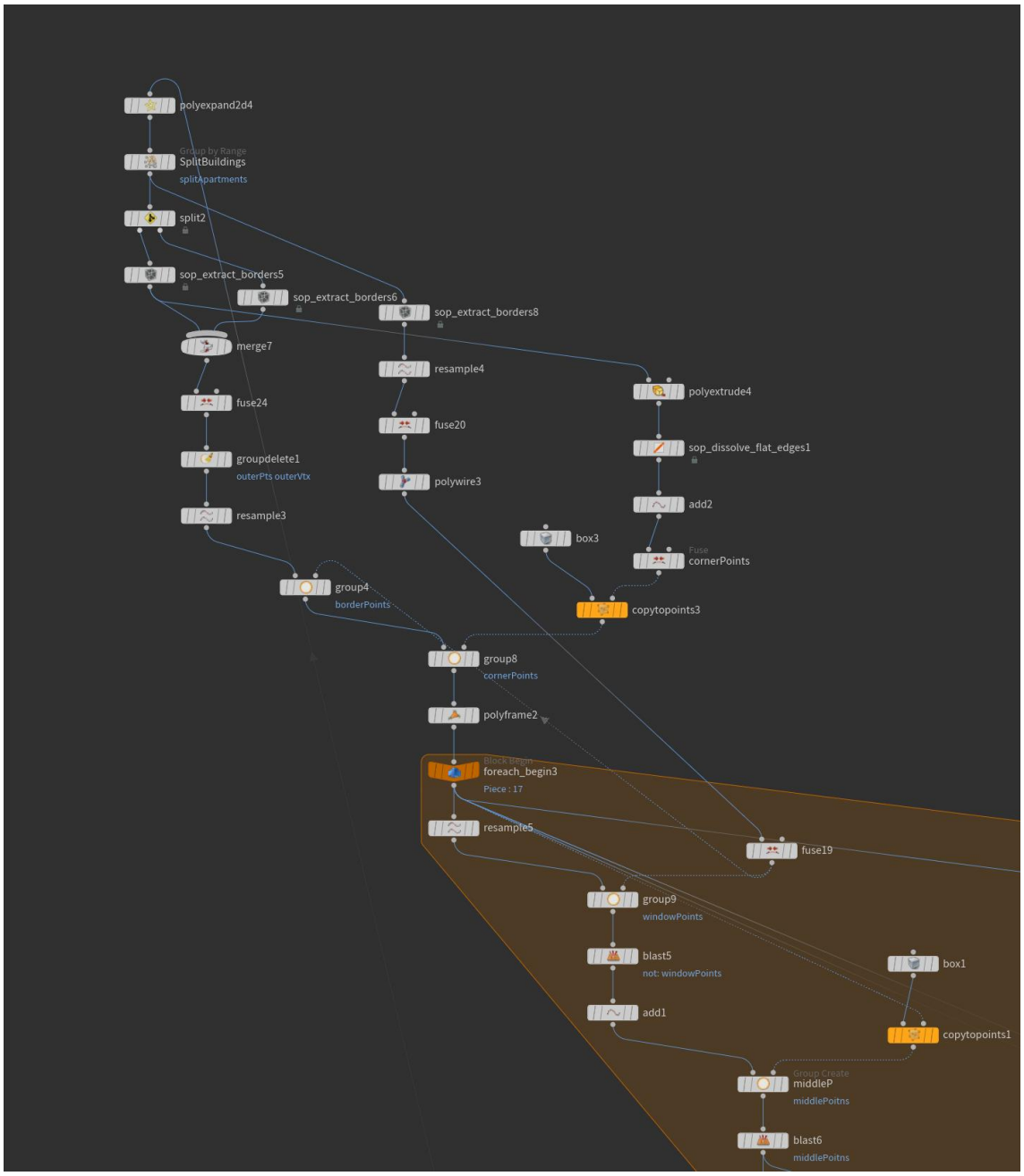


Image 55: Procedural apartment nodes pt1

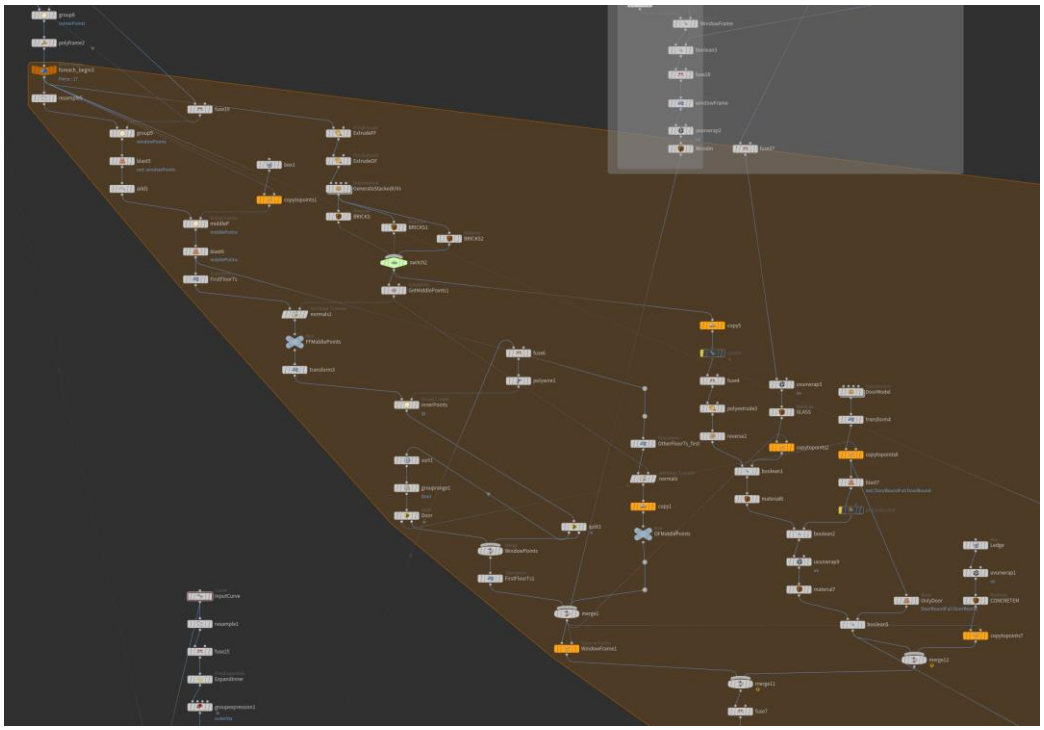


Image 56: Procedural apartment nodes pt2

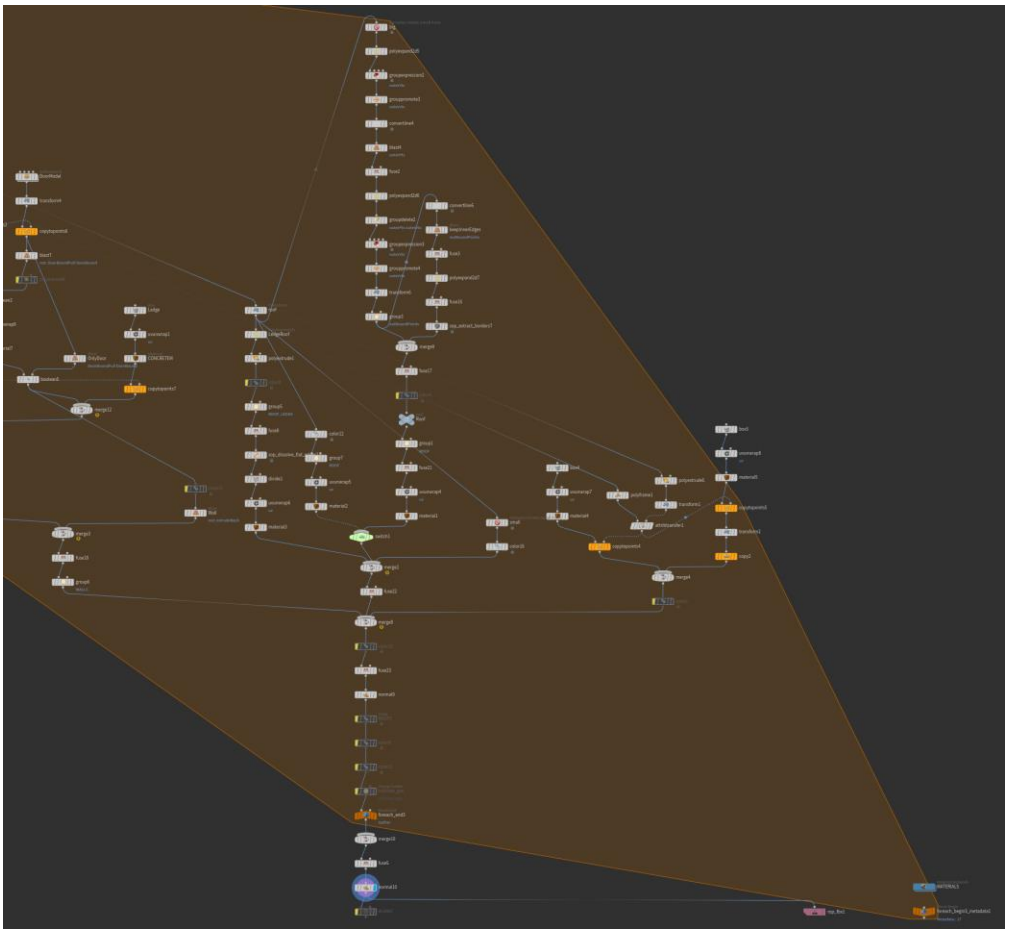


Image 57: Procedural apartment nodes pt3

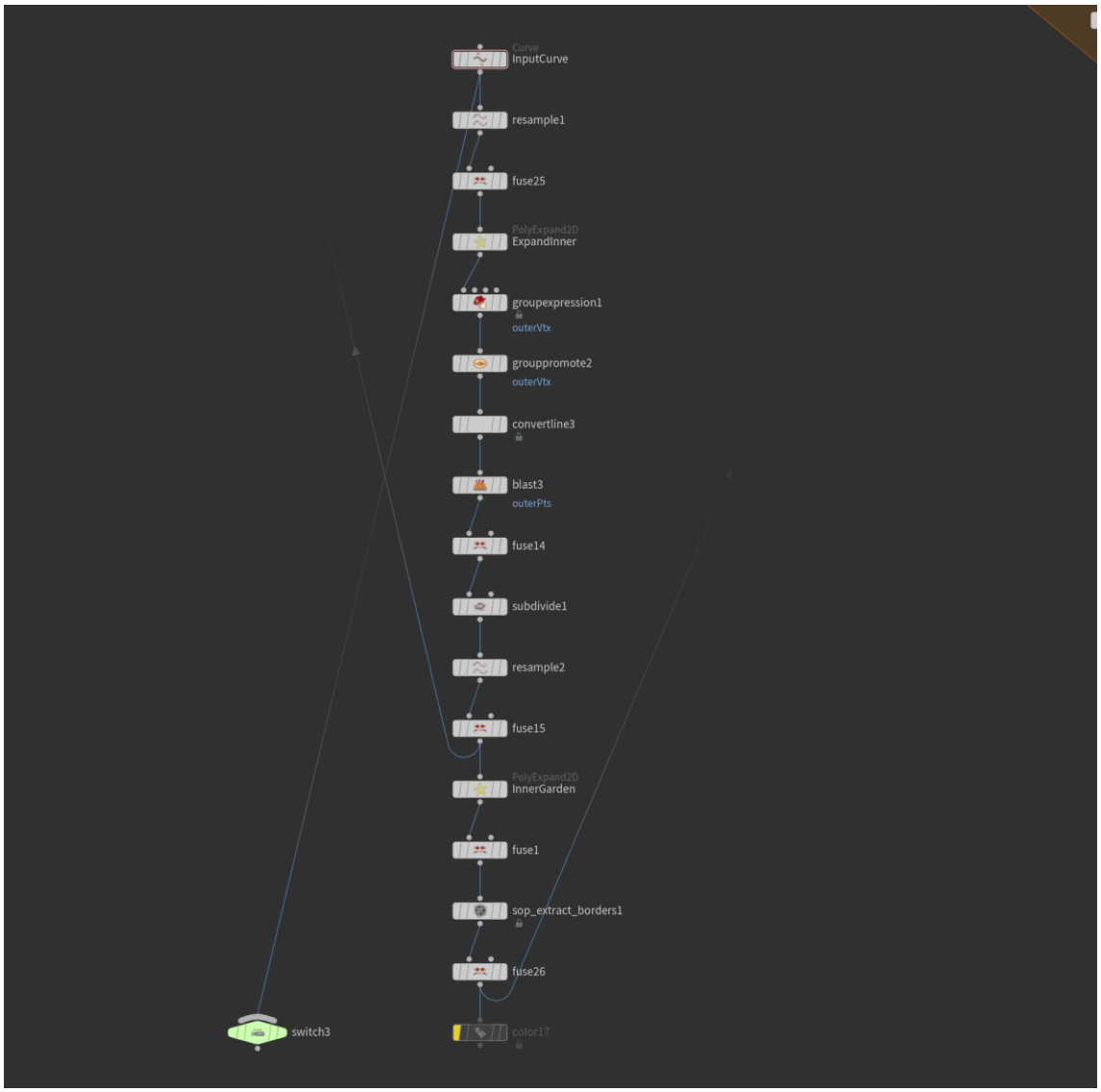


Image 58: Procedural apartment nodes pt3

A.3. PROCEDURAL SKYSCRAPER

This is how the skyscraper is generated from inside

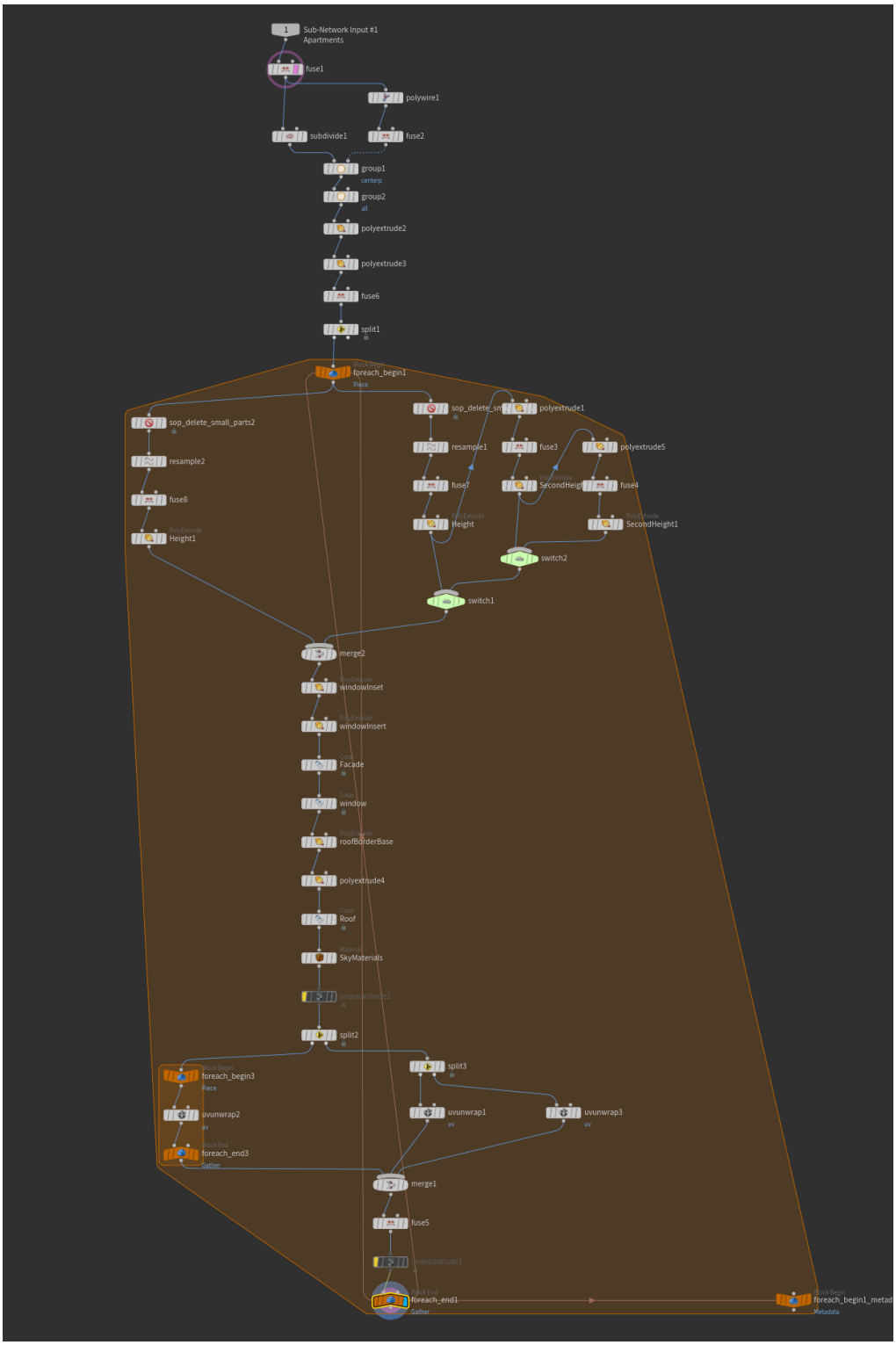


Image 59: Procedural skyscraper final node distribution

A.4. PROCEDURAL CITY

In this appendix a breakdown of the node composition for generating the city is shown by element.

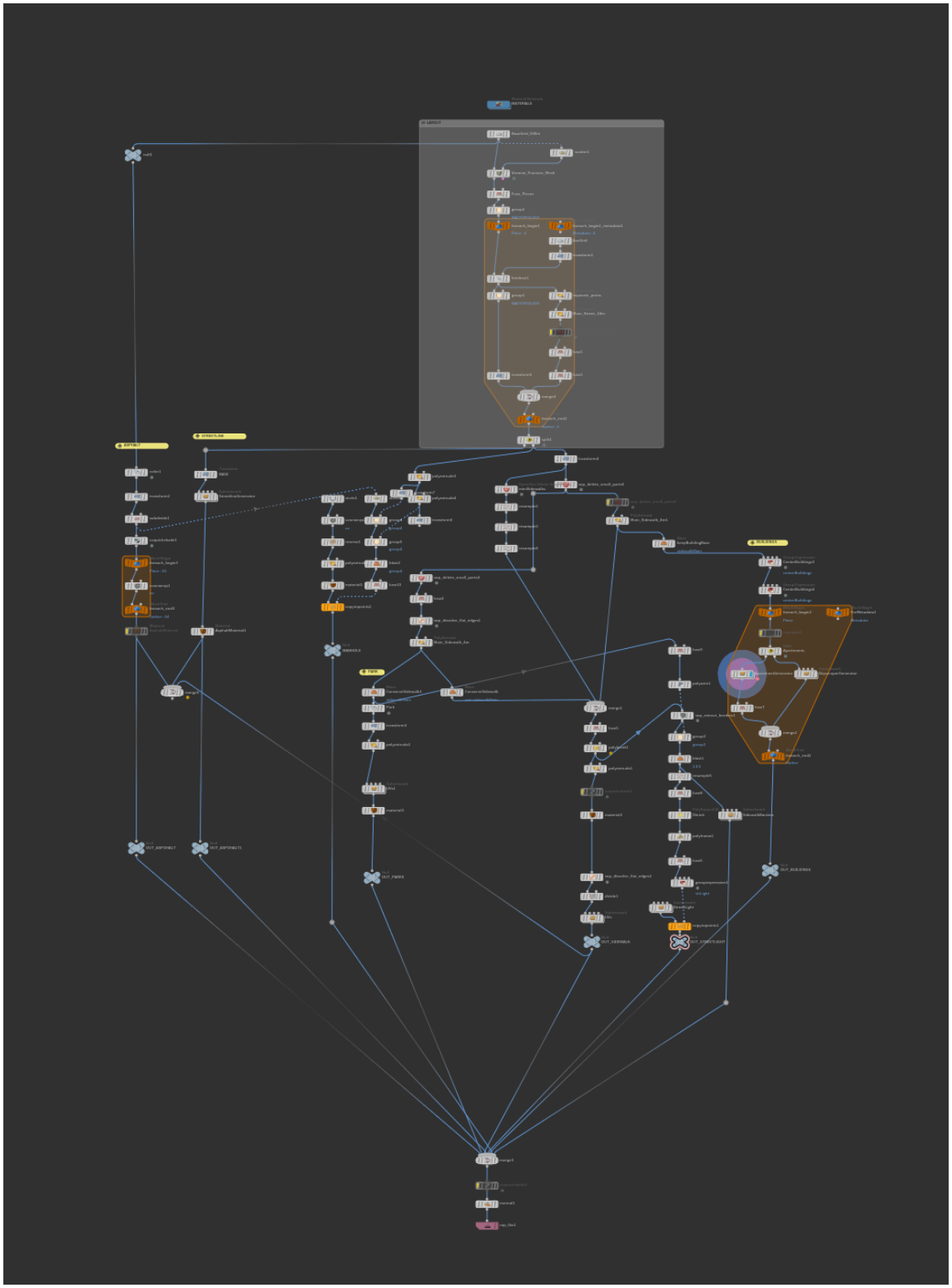


Image 60: Procedural city nodes full layout

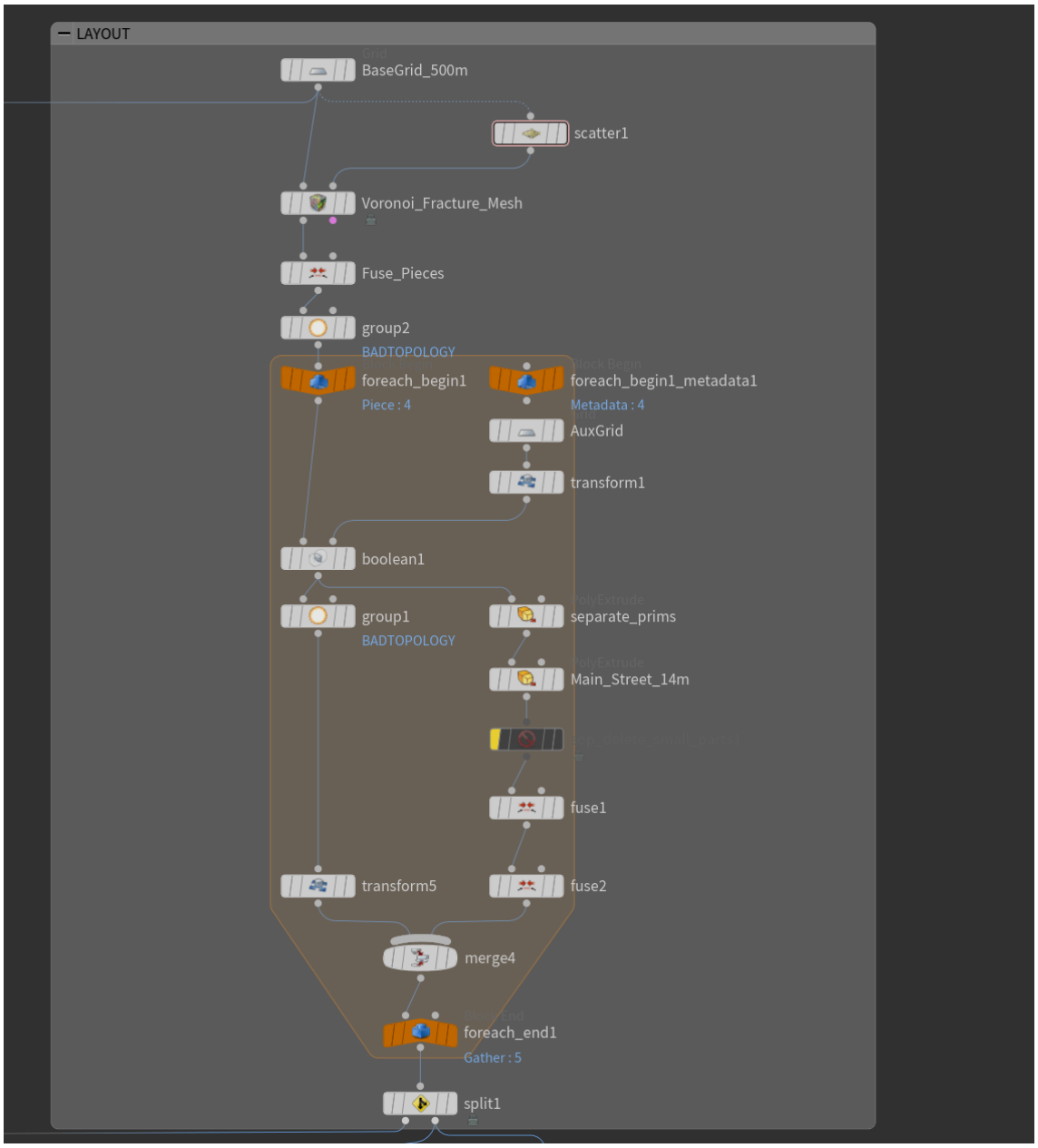


Image 61: Procedural city input nodes

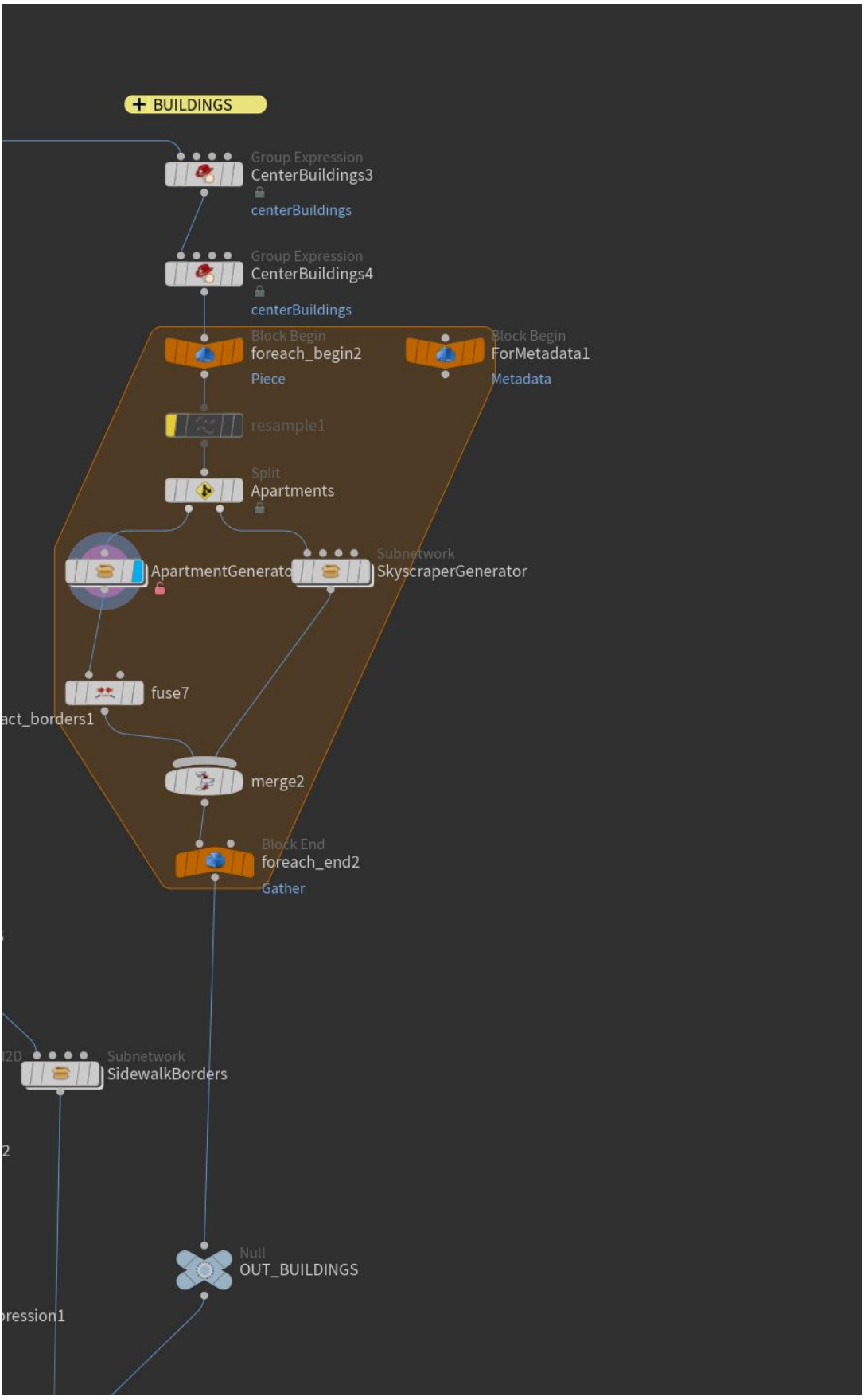


Image 62: Procedural city buildings nodes

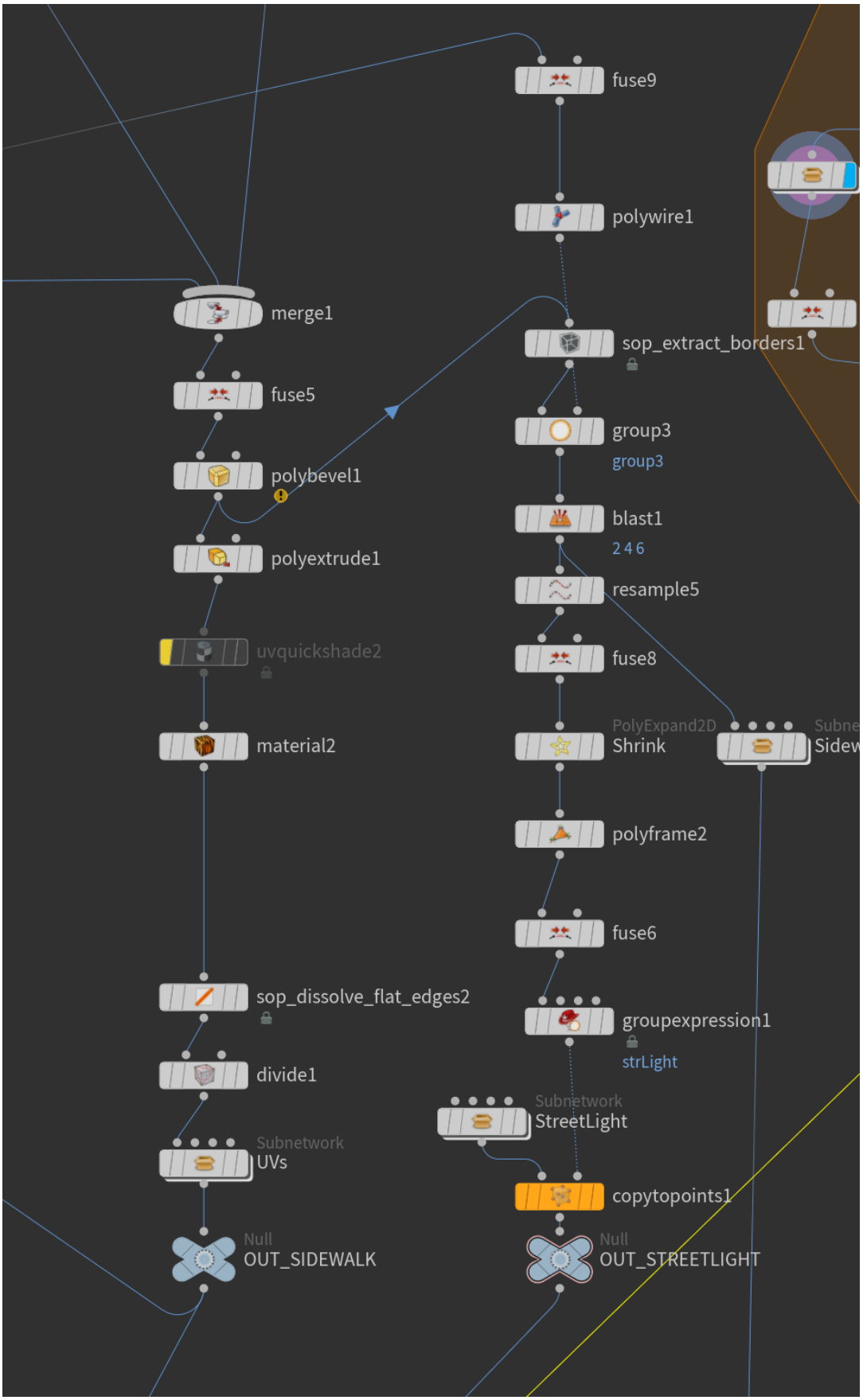


Image 63: Procedural city sidewalk and streetlight nodes

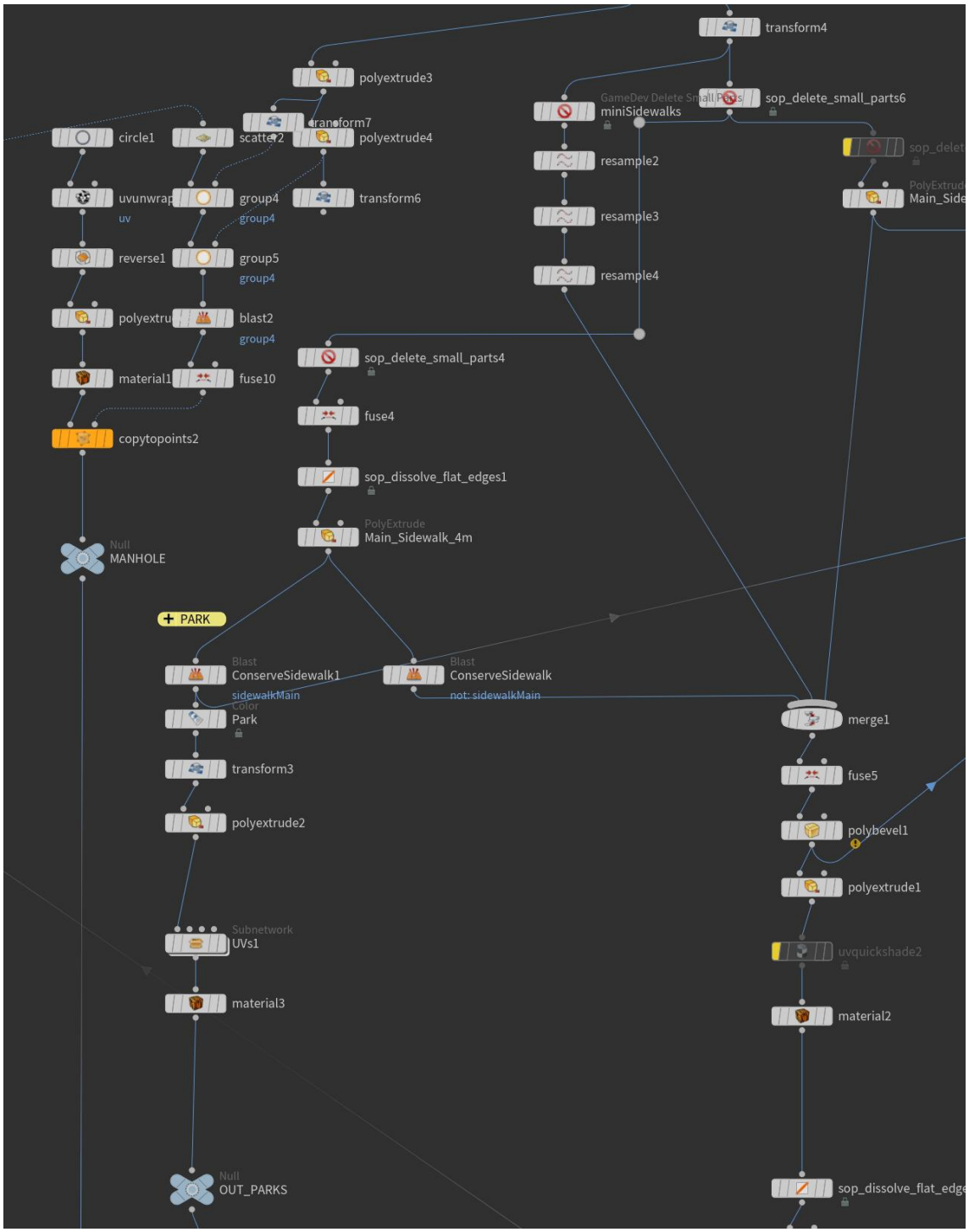


Image 64: Procedural manhole and parks

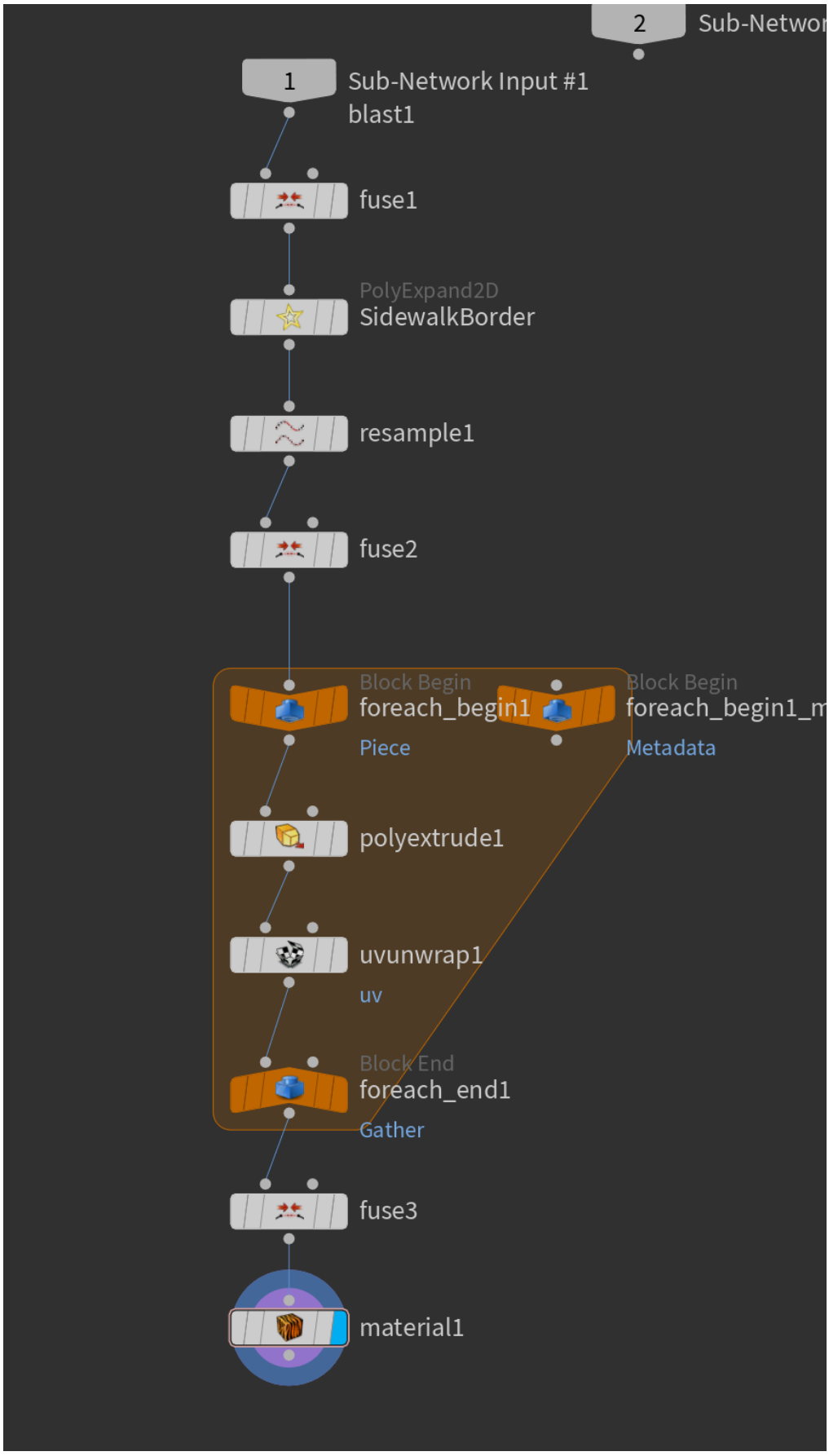


Image 65: Procedural city sidewalk border blocks

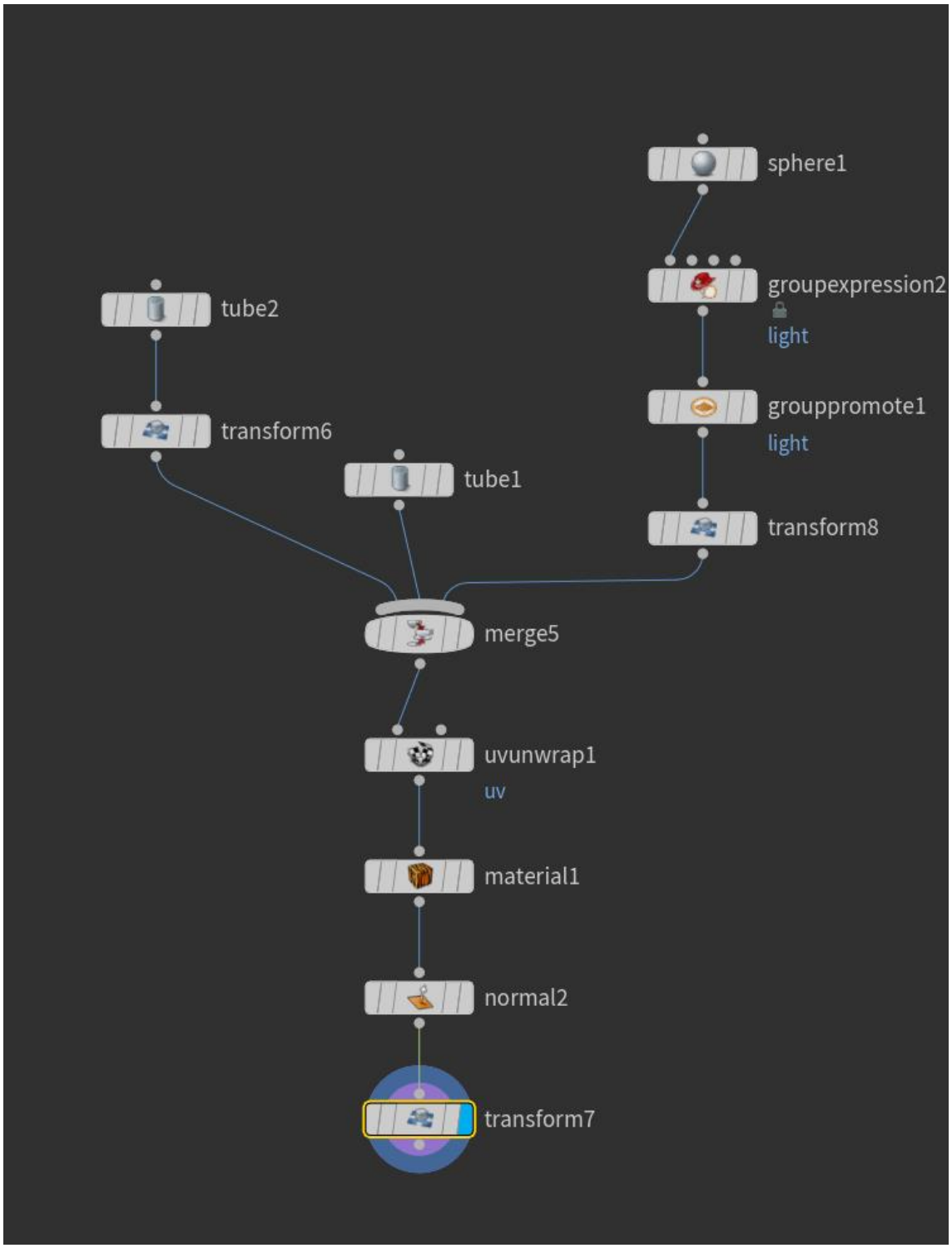


Image 66: Procedural city street light mesh

B PROJECT ACCESS

Here is a link to the project files.

https://drive.google.com/drive/folders/1onr-YOZKS_GwQ3dmYkee5wVEvyNoYKLX?usp=sharing